

Das API Handbuch

Ein Artikel über die Verwendung des APIs unter Visual Basic

Von Götz Reinecke

reinecke@activevb.de

Version 1.02

Copyright by <http://www.activevb.de> 7/2001

Nachdruck und Vervielfältigung jeder Art, auch auszugsweise, dies gilt ebenso für elektronische Medien, bedürfen der ausdrücklichen Genehmigung des Autors.



Vorwort

Das API ist schon recht betagt und seine Zeit als Windows-Schnittstelle läuft, für den Fall, daß sich die anstehende neue .net Philosophie in den nächsten Jahren durchsetzen sollte, einem absehbaren Ende entgegen. Dennoch wird es bis dahin Bestand haben und seine derzeit allgegenwärtige Präsenz nur langsam aufgeben. Bis dahin ist es fast unabdingbar sich mit ihm, auch unter VB, auseinander zu setzen. Deshalb findet ein Artikel wie dieser nach wie vor seine Berechtigung.

Auf der Internet-Seite www.ActiveVB.de ist die Anwendung des APIs unter VB eigentlich zentrales Thema, und ein Tutorial dieser Art ist unlängst mehr als überfällig. Fast ein Jahr ist seit dem ursprünglichen Entschluß ein solches zu verfassen vergangen. Es letztendlich zu schreiben hat nur wenige Tage in Anspruch genommen

Die meiste Arbeit steckt mehr im gedanklichen Vorspiel des Abwägens und Verwerfens der grundsätzlichen Strategie über den möglichen Aufbau und der Aussortierung von Themen. Die Schwierigkeit lag vor allem in der Entwicklung eines für jeden nachvollziehbaren roten Fadens. Und das war gar nicht so leicht, da das API im Umgang mit VB-Mitteln sich alles andere als unproblematisch darstellt. Ja, man kann fast schon sagen das eigentliche Arbeiten mit dem API besteht überwiegend aus Problembewältigungen. Aus diesem Grund werden auch eine ganze Menge der unterschiedlichsten Kniffe und Tricks benötigt, die leider nicht zwangsläufig in einem zusammenhängenden Kontext stehen und daher nicht einer ach so schönen aufbauenden Kontinuität folgen.

Zur letztendlichen Findung eines Konzepts hat das ausgezeichnete Tutorial von Paul Kuliniewicz auf www.vbapi.com nicht unwesentlich beigetragen. Es diene als wichtiger Anhaltspunkt und Referenz für die elementaren Eckpunkte, kleinere Auszüge daraus sind in diesem Artikel in frei übersetzter Form wiederzufinden. Ich möchte mich daher an dieser Stelle nochmals ausdrücklich für dessen Bereitstellung durch ein kräftiges virtuelles Händeschütteln bei ihm bedanken.

Ich hoffe das Tutorial ist ausreichend verständlich formuliert und ermöglicht Ihnen in Sachen API ein gutes Stück weiterzukommen. Es richtet sich in erster Linie sowohl an Einsteiger als auch den fortgeschrittenen Programmierer, vielleicht findet ja sogar der ein oder andere Profi darin einen ihm bis dato unbekanntem, brauchbaren Kniff. Wenn die Möglichkeit besteht, drucken Sie diese Seiten aus, das erleichtert das bei diesem Thema fast unvermeidliche Vor- und Zurückblättern. Anzumerken sei noch, daß passend zum Artikel insgesamt 27 Visual Basic Beispiele gehören, sie sollten sich beim Entpacken des vorliegenden Textes in ein eigenes Verzeichnis namens "Codes" extrahiert haben.

Wenn ich meine Arbeit gut gemacht habe und Sie das meiste des hier geschriebenen nachvollziehen können, sollten Sie im Anschluß daran über ein vernünftiges Rüstzeug im Umgang mit der API verfügen und in der Lage sein, weiterführende Literatur zum Thema problemlos zu verstehen. Also viel Vergnügen!

Inhalt

Vorwort	2	Strukturen respektive Typen und 64-Bit Integers	18
Was ist eigentlich ein API?	3	Die Zeichenfolge als Querulant	23
Welche Aufgaben übernimmt das Windows-API?	4	Handles	28
Wo ist das API denn zu finden?	5	Gerätekontexte [DeviceContexts]	30
Grundsätzliche Bestandteile des APIs	6	Flags	31
Für und Wider der Verwendung	7	Die SendMessage	33
Die Deklaration einer API-Funktion	8	Callbacks	37
Parameter	10	SafeArrays	38
Verschiedene Datentypen	11	Schnelle Bitmap-Operationen	45
Was Sind Pointer?	12	Information ist alles	49
ByVal und sein Kumpel ByRef	13	Subclassing, der geheime Nachrichtendienst	51
Zeigerspiele	15	Schlußwort	60
Boolean und Arrays	17		

Was ist eigentlich ein API?

Die Abkürzung API steht für "Application Programming Interface" und stellt eine Softwareschnittstelle dar. Bevor wir das näher klären, soll anhand einer kleinen Geschichte erstmal definiert werden, was eine Schnittstelle ist:

Frau Bastelmüller beschließt Ihr Eigenheim mit einem neuen und handgeschmiedeten Briefkasten zu verschönern und möchte hierfür gerne die Fertigkeiten der ortsansässigen Schlosserei "Dengel und Klopp" in Anspruch nehmen. Dazu sucht sie den kleinen, dem Betrieb angegliederten Laden auf und erläutert dem dortigen Angestellten, Herrn Pingelig, Ihre Wünsche. Der muß unsere aufgeregte Frau Bastelmüller erstmal beruhigen und ihre Vorstellungen über den neuen Briefkasten in eine handwerklich auch wirklich umsetzbare Form bringen. Es dauert nicht allzu lange und die beiden sind sich über Beschaffenheit, Abmaße, veranschlagte Arbeitszeit und den Preis des neuen Kastens einig. Herr Pingelig eilt daraufhin in die hinter dem Laden gelegene Werkstatt und erläutert dort dem Schlossermeister, Herrn Kloppdruff, den Auftrag der Frau Bastelmüller. Da Herr Pingelig und Herr Kloppdruff seit Jahren ein eingespieltes Team sind, bedarf es hier nicht vieler Worte und der Meister legt ohne Umschweife los. Einen Tag später ist der Kasten fertig und Herr Pingelig übergibt in seinem Laden der zufriedenen Frau Bastelmüller ihren neuen Briefkasten.

Hübsch, nicht wahr? Herr Pingelig bildet eben hier die besagte Schnittstelle zwischen der Kundin und dem Handwerker. Die Kundin hat einen bestimmten Auftrag, der eigentlich direkt an den Handwerker gerichtet ist. Der Verkäufer vermittelt zwischen diesen beiden Parteien. Und genau das macht auch das API, es bietet ebenso wie Herr Pingelig eine Vermittlung zwischen zwei sehr unterschiedlichen Gruppen an. Auf der einen Seite stehen wir als Anwendungsentwickler [Frau Bastelmüller] mit unserer Software und auf der anderen das jeweilige Betriebssystem [Herr Kloppdruff], in unserem Falle Windows. Wenn wir für unsere Anwendung eine bestimmte Funktionalität wünschen, richten wir uns also nicht direkt an das Betriebssystem, sondern bemühen einen Vermittler, das API, welches die geforderte Dienstleistung an die richtige, nie direkt in Erscheinung tretende Stelle des Systems weiterleitet. Auch das Resultat der dort verrichteten Arbeit erhalten wir niemals unmittelbar von dort, sondern auch wieder nur über den Vermittler, den API.

Fassen wir den Ablauf also kurz zusammen. Die Lösung einer Aufgabe unserer Anwendung erfordert die Nutzung einer bestimmten Funktionalität des Betriebssystems. Hierfür wenden wir uns an das API und teilen ihm die Eckdaten mit. Das API weiß welche Stellen im System hierfür verantwortlich sind, paßt unsere Daten entsprechend an und leitet die Aufgabe dorthin weiter. Nach getaner Arbeit liefert das System dem API das Ergebnis, welches wiederum in aufbereiteter Form von ihm an unsere Anwendung zurückgegeben wird.

Sie können davon ausgehen, daß jede Anwendung unter Windows exzessiven Gebrauch des APIs macht. Selbst wenn explizit kein Aufruf des APIs in einem Programm auftaucht und dem Anschein nach, nur die grundlegenden Befehle einer Programmiersprache ausgeführt werden, steckt hinter jedem dieser eine Reihe von Aufrufen des APIs.

Jedes Betriebssystem besitzt ein API. Windows, Linux und sogar der Macintosh verfügt über eine solche Schnittstelle zum System. Das ist aber auch schon die einzige Gemeinsamkeit. Zergliedert man das jeweilige API in grobe Funktionsblöcke, lassen sich vielleicht noch prinzipielle Übereinstimmungen feststellen, im Detail allerdings ist ihr Handling vollkommen verschieden. Schon deshalb ist es grundsätzlich unmöglich eine unter Windows erstellte Software auf einem Linux-Rechner laufen zu lassen, da die konkreten Verknüpfungstellen jeweils komplett anderes aussehen. Eine Ausnahme bietet sich nur unter Verwendung eines Emulators, der im wesentlichen aber auch nichts anderes tut, als zwischen den beiden unterschiedlichen APIs zu übersetzen.

Das API ist die Basis jeder höheren Programmiersprache. In solchen erledigt das laufende Programme nicht sämtliche Aufgaben selber, sondern beauftragt damit oftmals andere Programme. Insbesondere bedeutet dies, daß es recht häufig Aufgaben und Funktionen an das jeweilige Betriebssystem weiter delegiert.

Um z.B. Daten auf eine Diskette zu schreiben, wird man das eigene Programm nicht derart gestalten, daß es die benötigten Operationen bitweise angepaßt und entsprechend dem vorhandenen Dateisystem mühevoll selbst ausführt. Vielmehr greift man auf fertige Funktionseinheiten des Betriebssystems zurück, die es gestatten durch Benennung und Parametrierung die Aktion wie gewünscht durchzuführen.

Der Vorteil dieser Vorgehensweise ist offensichtlich. Der Anwendungsentwickler muß sich nicht in jedem einzelnen Programm akribisch um essentielle Details wie Dateizugriffe, Speicherverwaltung, Zeichenwerkzeuge etc. kümmern. Das spart Zeit und schont den Speicherbedarf. Zudem ist davon auszugehen, daß die vom Betriebssystem dargebotenen Funktionen bereits stark optimiert sind und sich über einen längeren Zeitraum bewährt haben.

Sollten sich bedingt durch fortschreitende Entwicklungen in Sachen Hardware, Verbesserungen oder gar Änderungen in deren Programmierung ergeben, so braucht der Softwareentwickler nicht in all seinen ehemals geschriebenen Programmen darauf einzugehen, sondern kann diese Aufgabe getrost einem Update des jeweiligen Systems überlassen. Auch der Aspekt eines konformen Designs bzw. einer gleichbleibenden Oberfläche und eines konstanten Handlings erweist sich für den letztendlichen Nutzer einer Software als äußerst hilfreich.

Welche Aufgaben übernimmt das Windows-API?

Im wesentlichen steht in der Windows-API alles zur Verfügung, was das eigentliche und allseits bekannte Aussehen und Feeling von Windows ausmacht. Im Kern umfaßt es obengenannten Dinge wie Dateizugriff, Druckersteuerung, Zeichenwerkzeuge, Internethandling etc. als auch die Windows-typischen Elemente für einen konformen Umgang. Hierzu gehört zum Beispiel der CommonDialog [Öffnen, Speichern unter, Schriftart wählen etc.], ListBoxen, ComboBoxen, TextBoxen, Systemeinstellungen, die Windows-Shell, und natürlich die Fenster selber.

Bildlich vorstellen läßt sich in diesem Zusammenhang der Begriff Schnittstelle wohl am besten als ein Drei-Schichten-Modell. In der untersten befinden sich die transparenten, also für den Programmierer nicht sichtbaren und direkt nutzbaren, Funktionalitäten des Betriebssystems. Dazu gehören Dinge wie Taskmanaging, Fensterverwaltung, Treiber etc. Die erste und oberste Schicht bildet die eigentliche Software die wir alltäglich entwickeln. Da aber zwischen diesen beiden zwangsläufig eine Kommunikation stattfinden muß, gibt es als "Übersetzer" das API, das oftmals eben nur reine Vermittlungsdienste leistet. Diese Zwischenschicht ist also unabdingbar und bietet eine überraschend große Fülle an Möglichkeiten für den alltäglichen Gebrauch.

Unter Windows laufende Programme machen, wie bereits Eingangs angeschnitten, intensiven Gebrauch vom API. Selbst wenn, wie z.B. in VB, ein Programm scheinbar auch vollkommen ohne das API auskommt, nehmen die diversen Befehle, versteckt im Hintergrund, überwiegend das API in Anspruch, um ihre zugeordnete Funktionalität erledigen zu können. Als Beispiel sei nur das gerühmte Event-Modell von VB benannt. Sämtliche dort zur Verfügung stehenden Ereignisse werden vom API ausgelöst und unter Verwendung desselben auch wieder abgearbeitet.

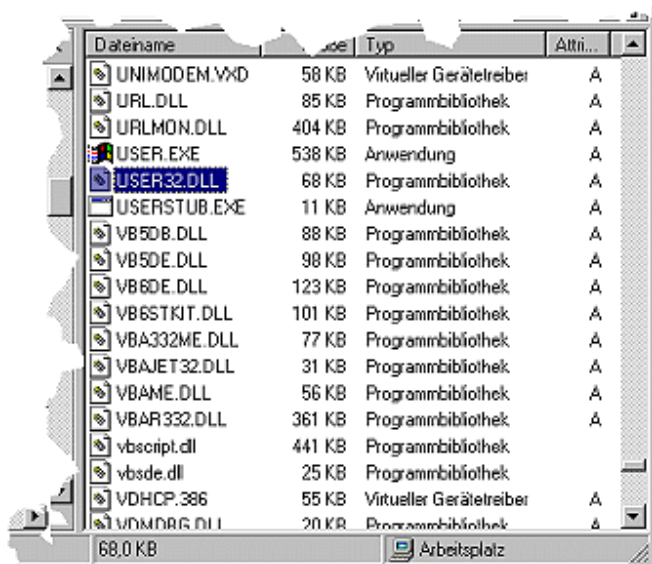
Wenn wir unter VB ein Formular mit einem CommandButton versehen und dessen Clickereignis für das Aufrufen einer MessageBox nutzen, haben wir bereits schon eine lange Liste von API-Funktionen in unser Programm implementiert. Sie sind für uns nicht sichtbar, da VB sie in Klassen gekapselt hat und somit unserem kritischen Blick entzieht. Trotzdem sind sie allgegenwärtig und das in jeder Programmiersprache, ja selbst in Assembler finden sie ihre Anwendung. Es spricht also nichts dagegen die Nutzung des APIs unter VB lediglich der transparenten, vorgegebenen Klassenkapselung zu überlassen. Vielmehr sollten wir dieses mächtige Werkzeug einsetzen um die vorgegebenen Unzulänglichkeiten der Klassifizierung auszubessern oder zu erweitern.

Wo ist das API denn zu finden?

Ausnahmslos jede Funktion des APIs steht in einer Datei mit der Endung .dll, welche sich wiederum überwiegend im Windows-Systemverzeichnis tummeln. Sie können aber auch in anderen durch die Umgebungsvariable PATH beschriebenen Verzeichnissen oder im Ordner einer Anwendung selbst abgelegt sein.

DLL ist die Abkürzung für "Dynamic Link Library", es handelt sich hierbei um nicht direkt ausführbare Dateien die eine Sammlung von Funktionen beherbergen. Diese Funktionen können an externe Programme "exportiert", sprich von diesen eingebunden werden. Die DLL steht dabei nur ein einziges mal im Speicher und wird bei weiterem Bedarf, falls nicht bereit schon geladen, einfach höher referenziert.

Bekanntestes Beispiel dürfte wohl die VB Laufzeitbibliothek sein. Sie ist ebenfalls eine DLL, die aber nur einmal vorhanden sein muß und von beliebig vielen VB-Programmen gleichzeitig genutzt werden kann



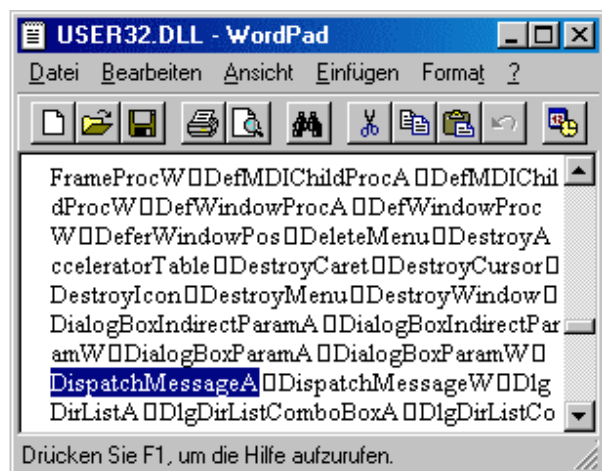
Dieser praktische Aufbau gestattet windowsbasierten Programmen einen leichten und schnellen Zugriff auf das API. Die Funktionen sind oftmals nach Themenschwerpunkten in je einer DLL zusammengefaßt. Die am häufigst vertretenen sind hierbei:

- **User32.dll** Allgemeine Benutzerfunktionen, Nachrichtenverwaltung, Windowselemente, Timer
- **Kernel32.dll** Betriebssystembezogene Funktionen wie Speicherhandling, Systeminformationen
- **Gdi32.dll** Zeichenwerkzeuge, Gerätekontext bezogene Operationen
- **Advapi32.dll** Nutzung der Windows Registry, Nutzerrechte, Sicherheit und EventLogging
- **Wininet.dll** Internet-Funktionen des Internet Explorers
- **Winspool.dr** Druckerhandling
- **Wsock32.dll** Netzwerk und rechnerübergreifende Kommunikation
- **Shell32.dll** Dateifunktionen, Dateicons, starten von Anwendungen, Dateidialoge

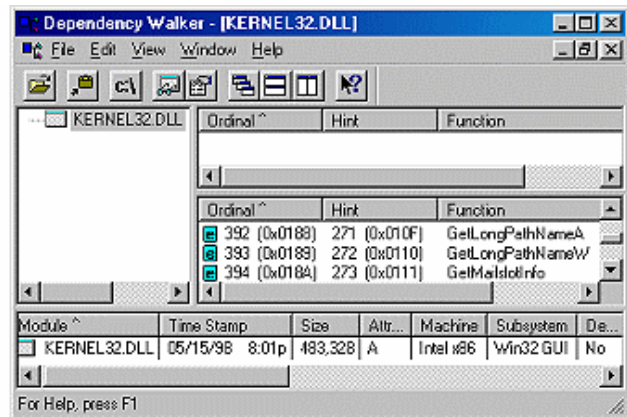
Woher sind nun die entscheidenden Informationen nehmen, welche Funktionen sich in einer zum Beispiel unbekanntes, also nicht in irgendeinem Kompendien nachschlagbaren, DLL befinden?

Dies ist gar nicht so einfach herauszufinden. Eine etwas brachiale aber oft erfolgreiche Methode ist das Öffnen der DLL in einem Texteditor, wie zum Beispiel WordPad. Die Funktionsnamen befinden sich in der als ASCII-Text geöffneten Datei meist am Anfang des letzten Drittels.

Probieren Sie dies ruhig einmal bei den verschiedenen DLLs im Systemverzeichnis aus. Meist, aber nicht immer, klappt es auf diese Art und Weise. Um sich eine grobe Übersicht zu verschaffen reicht dies oftmals schon aus. Anhand der gefundenen Funktionsnamen läßt sich dann weiter in der MSDN oder im Internet nach einem Beispiel oder einer Beschreibung recherchieren.



Professioneller hingegen erscheint das Tool **Dependency Walker**. Hiermit ist eine Fülle von Informationen über die jeweilige DLL zu erfahren. Neben den eigentlichen Funktionsnamen, wird auch über die Ordinalzahl sowie die Einsprungadresse der einzelnen Funktionen Auskunft gegeben. Zudem gibt es allgemeine Einblicke in die DLL-Version, das Erstellungsdatum, die Größe etc.



Besonders interessant, ist die Möglichkeit, und das ist sicherlich auch das eigentliche Arbeitsfeld dieses Programms, sich Abhängigkeiten anzeigen zu lassen. So kann es vorkommen, daß Funktionen in DLLs auf Funktionen in anderen DLLs verweisen. Dies ist besonders wichtig, um bei Weitergabe der eigenen Anwendung vor Überraschungen in Form von fehlenden Komponenten gefeit zu sein.

Grundsätzliche Bestandteile des APIs

Überwiegend sind in ihm zwar Funktionen vertreten, darüber hinaus gibt es aber auch noch eine Reihe anderer Bestandteile die Beachtung finden sollten, die folgende Aufzählung verschafft uns einen Überblick. Auf die hier genannten verschiedenen Komponenten, wird zu einem späteren Zeitpunkt nochmals detaillierter eingegangen, daher ist es jetzt nicht zwingend notwendig das genannte vollkommen zu verstehen:

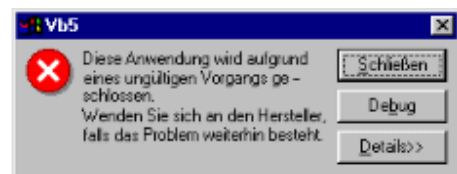
- **Funktionen:** Wie bereits erwähnt, bilden sie den eigentlichen Kern und bewerkstelligen sämtliche Aufgaben. Oftmals liegen sie in der DLL nur als Hüllfunktion vor, von der aus eine oder mehrere weitere betriebsysteminterne Funktionen aufgerufen werden, nicht zwingend ist dort tatsächlicher prozeduraler Code vorhanden. Die Funktionen sind themenorientiert in DLL-Dateien in einer Art Bibliothek angesammelt und gespeichert.
- **Makros:** Ein *Makro* ist eine einzelne API-Funktion, in der mehrere andere API-Funktionen zu einer speziellen Funktionseinheit vereinigt worden sind. Sie stellen sich nach außen hin aber wie je andere gängige API-Funktion dar und werden auch wie eine solche gehandhabt. *Makros* sind eher selten anzutreffen.
- **Subs:** Sind Funktionen ohne Rückgabewerte. Ansonsten entsprechen sie dem bereits oben gesagten. Auch *Subs* findet man nicht sehr häufig.
- **Callbacks:** Begrifflich sind sie das Gegenstück zu einer API-Funktion. Eine *Callback* befindet sich ausschließlich in einem Anwendungsprogramm und wird vom Betriebssystem beim Zustandekommen bestimmter Bedingungen aufgerufen. Im wesentlichen kann dies zum einen beim Eintreffen eines externen ausgelösten Ereignisses stattfinden oder zum anderen bei einer Enumeration Vorzug finden. Durch diese Technik wird einer Anwendung oftmals eine hohe Dynamik im Wechselspiel mit dem Betriebssystem zuteil. Im Falle einer Enumeration [Aufzählung] ruft das System für jeden Eintrag einzeln der Reihe nach die *Callback* auf, bis die Liste ihr Ende erreicht hat oder aber von der *Callback* ein eindeutiger Wert, wie z.B. 0 zurückgegeben wird. Zur Definition einer *Callback* wird die Adresse der vom Betriebssystem aufzurufenden Funktion an eine API-Funktion übergeben. Ab VB5 ist die Ermittlung der Adresse mit Hilfe des *AddressOf* Operators möglich geworden.

- **Konstanten:** Die Windows-API erwartet für ihre verschiedene Funktionen oftmals sogenannte Konstanten diese nehmen in Form von *Flags* oder Kommandos direkten Einfluß auf den weiteren Verlauf und die wesentliche Funktionsweise. Konstanten besitzen überwiegend einen numerischen Wert. Damit der Umgang mit diesen Zahlen nicht zu unübersichtlich wird, hat man sich darauf geeinigt ihnen aussagekräftige Namen zu geben. Dieses müssen ebenfalls wie die *Strukturen* im eigenen Programm vorab definiert werden.
- **Nachrichten, Messages:** Nachrichten, geläufiger unter der Bezeichnung Messages, sind eine Sonderform der Konstanten. Ihre Definition gestaltet sich zwar genauso wie ihre Kollegen, sie werden aber letztendlich dazu genutzt, Ereignisse und Eigenschaften in Objekten auszulösen bzw. zu ändern. Nachrichten heben sich zum besseren Verständnis für den Entwickler durch einen Präfix ab, der die Konstante mit dem betroffenen Objekttypen, assoziieren lassen soll. [LB_ für ListBoxen, CB_ für ComboBoxen etc.]
- **Flags:** zu deutsch Flaggen, sind ebenfalls eine Untergruppe der Konstanten. Sie zeichnen sich durch ein als numerischer Wert vorliegendes Bitmuster aus und tauchen in thematisch geordneten Gruppen auf. Jedes Flag einer Gruppe besitzt einen unverwechselbares, eindeutiges Bit welches für einen ganz bestimmten Zustand steht. Flags lassen sich auf Grund ihrer Eindeutigkeit miteinander kombinieren und bilden dadurch einen einzelnen 32-Bit langen Steuerwert.
- **Strukturen:** Die Bezeichnung *Struktur* rührt aus der C-Programmierung, ist unter VB eher unter dem Namen *Typ* bekannt und stellt einen benutzerdefinierten Variablentyp dar. Sie können verschiedene weitere Variablen und auch Typen in einer Variablen zusammenfassen. Diese Einzelteile werden im C-Jargon auch *Members* genannt. Viele API-Funktionen erwarten als Übergabeparameter solche Strukturen, denn sie besitzen den Vorteil recht einfach eine Fülle von Informationen auf einen Schlag übergeben zu können. Die Deklaration dieser Typen, also das Konstrukt, steht nicht öffentlich zur Verfügung und muß von daher in VB selbst vorgenommen werden.

Für und Wider der Verwendung

Die Verwendung des APIs hat auch seine Tücken. Seltsamerweise ist der Aufruf von API-Funktionen fehleranfälliger, als ein äquivalenter Befehl der jeweils verwendeten Programmiersprache. Das beruht auf folgendem Umstand: Die gesamte API-Bibliothek wurde in C++ entwickelt und auch auf diese Sprache ausgelegt, so daß die Variablen und die Parameterübergabe leider immer der C-Syntax anzupassen sind. Nicht selten aber kommt es gerade hierbei zu Problemen. C arbeitet mit Variablen und Techniken die in VB nicht standardmäßig implementiert sind. Dazu gehört zum Beispiel die Verwendung von Zeigern, nullterminierten Strings und anderen bösen Dingen, die der VB-Programmierer normalerweise nicht benötigt und daher auch nicht kennt. Daher gilt unser Hauptaugenmerk im Umgang mit dem API auch genau diesen vielen kleinen Anpassungen, Konvertierungen und Sonderregelungen die die Unterschiede zur C-Syntax ausbügeln sollen.

Ein weitere unangenehme Nebenwirkung, besteht darin, daß API-Funktionen dazu neigen ihr Scheitern recht spektakulär kundzutun. Anstatt einen netten kleinen Fehlerdialog einzuspielen, fällt es ihnen nicht sonderlich schwer das aufrufende Programm, gegebenenfalls inklusive IDE mit dem Hinweis einer allgemeinen Schutzverletzung abrupt zu beenden. Diese Meldung ist oftmals das einzige Feedback das sie im Fehlerfall von der API zu Gesicht bekommen.



Ein weiteres Manko besteht darin, daß Visual Basic ohne jede API-Dokumentation daher kommt. Daher müssen ersteinmal Quellen gefunden werden, die Aufschluß darüber geben Funktionen in einer DLL oder auch allgemein überhaupt verfügbar sind, ganz zu schweigen von der Art ihrer Deklaration und Anwendung.

Trotzdem sollte das API durchaus einen guten Teil Beachtung an finden: API-Funktionen sind häufig bedeutsam mächtiger als ihre Kollegen, die Befehle, aus der jeweiligen Programmiersprache, das gilt ganz besonders für Visual Basic.

Dort kann zum Beispiel mit Hilfe der Caption-Eigenschaft die Titelzeile eines jeden Fensters, daß sich in der selben Anwendung befindet beliebig geändert werden. Was ist aber mit Titelleisten außerhalb der eigenen Anwendung? Die Lösung ist recht leicht, da es speziell hierfür eigens eine API-Funktion, namens `SetWindowText` daher kommt.

Ein weiteres Beispiel: Wenn in einer Anwendung der "Öffnen mit"-Dialog verwendet werden soll, kann dies ohne weiteres mit Hilfe des `CommonDialog-Controls` geschehen. Die Einbindung eines solchen Steuerelements hat aber den Nachteil, daß die Komponente bei der Weitergabe des Programms mitverteilt werden muß. Da das besagte OCX letztendlich aber nur eine einfache Kapselung der darunterliegenden API-Funktionen darstellt, sprich überwiegend lediglich aus knappen Hüllfunktionen besteht, ist es fast schon ratsam die tatsächlich für die Anwendung benötigten Hüllfunktionen selbst zu verfassen und somit das Programm vom Ballast des `Controls` zu entlasten.

Weiterhin ist zu beachten, daß viele Funktionalitäten wie zum Beispiel das allumfassende Arbeiten mit der Registry, oder die Nutzung externer Geräte wie Joysticks, Paddle oder WebCams in VB überhaupt erst durch das API nutzbar werden, da VB hier keine Möglichkeiten bereithält.

Abschließend läßt sich vielleicht folgende Prämisse aufstellen: Wo sich die Möglichkeit bietet bestimmte Aufgaben direkt aus VB heraus zu erledigen, sollte auch VB durchaus gegenüber dem API der Vorzug gewährt werden. Dies begründet sich zum einen in der leichteren Handhabbarkeit als auch in einer verlässlicheren Methodik. Wenn aber dem Programm mehr Geschwindigkeit und Dynamik gegeben werden soll, die Visual Basic nicht ohne weiteres unterstützt, ist das API die einzig mögliche Lösung.

Wir sind eigentlich von Haus aus daran gewöhnt mit der Kapselung des APIs in Form von Standard – Befehlen, -Methoden und –Eigenschaften ohne es zu bemerken umzugehen. Unter C++ hingegen hat man sich auf Grund des nicht zu unterschätzenden höheren Zeitaufwandes bei der doch recht umständlichen API-Programmierung entschlossen, deren Funktionen nachträglich in eine Klasse, die allseits bekannte MFC, zu isolieren. Daher gehen wir als Visual Basic Programmierer hier den umgekehrten Weg.

Die Deklaration einer API-Funktion

Auf den ersten Blick sieht die Verwendung des APIs unter Visual recht einfach und überschaubar aus. Im Prinzip ist es das auch. Bevor eine erstmalig Funktion genutzt werden kann, ist sie zunächst zu im oberen Abschnitt eines Formulars oder Moduls zu deklarieren. Mit der Deklaration wird Visual Basic gezeigt, wie die Funktion heißt, in welcher DLL sie zu finden ist, welche Parameter und welcher Rückgabewert erwartet wird. Ist dies geschehen, kann sie wie jede andere Visual BASIC-Funktion aufgerufen werden.

Eine Deklaration folgt immer dem selben untenstehenden Schema, das Token `Declare` am Anfang der Definition zeigt an das eine API-Funktion definiert wird. Die Definition muß grundsätzlich zu Beginn eines Modules oder eines Forms stattfinden. In einem Formular angewandt, kann die Funktion nur vom Type `Private` sein, was den Nachteil hat das der Gültigkeitsbereich der Funktion auf eben nur dieses eine Form beschränkt ist. Anders in Modulen, hier sind sowohl `Private` als auch `Public` Anweisungen gestattet. Mit letzterer ist die Funktion dann im ganzen Projekt verfügbar.

Die Syntax einer typischen Deklaration sieht wie folgt aus:

```
[{Public | Private}] Declare Function Funktions_Name Lib "DLL_Dateiname" [Alias "Funktions_Alias"] (Parameter) As Rückgabetyt
```


Da das ziemlich furchterregend aussieht, hier eine detailliertere Beschreibung:

- ***Funktions_Name*** Der Name der API-Funktion. Unter diesem ist die Funktion unter Visual Basic aufrufbar. Theoretisch kann dies bei korrekter Verwendung der Alias Anweisung ein beliebiger sein. Praktisch sollte aber hiervon abgesehen werden, da hierdurch ein Programm für andere schwerer lesbar wird, da die Funktion ja unter dem selbst kreierten Namen nicht jedem bekannt sein dürfte. Die vom API angebotenen Funktionsnamen sind meist auch aussagekräftig genug, so daß es nicht sehr schwer fällt deren eigentliche Funktionalität zu erahnen. Für den Fall, daß Sie das Schlüsselwort **Alias** nicht verwenden müssen, ist hier peinlichst genau auf die richtige Schreibweise des Funktionsnamens zu achten. Dieser ist case-sensitiv, d.h. es wird zwischen Groß- und Kleinschreibung unterschieden [name <> Name], sollte es hier versehentlich zu Abweichungen kommen, wird die folgende Fehlermeldung eingeblendet: "DLL-Einsprungpunkt '*Funktions_Name*' in '*DLL_Dateiname*' nicht gefunden".
- ***DLL_Dateiname*** Der Dateiname der DLL in der sich die Funktion befindet. Der vollständige Pfad kann, muß, bzw. sollte aber nicht, mit angegeben werden, da das Systemverzeichnis sich nicht immer zwingend an der selben Position steht. Weiterhin kann auch darauf verzichtet werden die Endung ".dll" anzugeben, dies wird empfohlen für die user32-, kernel32- und gdi32-Dll , für aller anderen sollte von dieser Option kein Gebrauch gemacht werden, da es Bibliotheken wie z.B. die winspool.drv gibt, die es bevorzugen nur über ihre eindeutige Endung angesprochen zu werden.
- ***Funktions_Alias*** Diese Anweisung ist überwiegend optional. Bei Verwendung muß hier der echte Name der Funktion, so wie er in der DLL verzeichnet ist, folgen. Dieses Token findet besonders bei Stringfunktionen seine Anwendung, da es von jeder Stringfunktion zwei Versionen, je eine ANSI- und eine Unicode-Version, gibt. Der eigentliche Funktionsname ist hier gleich, unterschieden werden sie durch das großgeschriebene Anhängsel A [für ANSI] bzw. W [für Wide Char = Unicode]. Für die Funktion **CompareString** ergäben sich folgende Varianten: CompareStringA und CompareStringW. Das ganz wirkt etwas verwirrend. Wir befinden uns nämlich gerade in einer Übergangsphase von ANSI zu Unicode. ANSI ist die ältere Ausgabe und genügt den heutigen Ansprüchen nicht mehr. Unicode bietet den Vorteil sämtliche Zeichen aller Sprachen in einem Zeichensatz vereinen zu können. Dies wurde erst durch Verwendung von 16-Bit für ein Zeichen möglich, so daß ein Unicode-Zeichen exakt zwei ANSI-Zeichen [8-Bit] entspricht.

Der Einsatz dies Schlüsselwortes **Alias** kann auch erforderlich werden, falls der eigentliche Name der Funktion bereits von einem VB-eigenen Befehl belegt ist oder eine Zeichenfolge darstellt, die in VB als Funktionsname syntaktisch nicht zulässig ist, wie beispielsweise ein vorangestellter Unterstrich [*_Funktions_Name*]. Ansonsten muß auch hier wieder der Name der Funktion exakt der Anforderung der DLL entsprechen. Es gelten dabei die selben Bedingungen und Hinweise in Sachen Schreibweise, wie bereits unter dem Punkt ***Funktions_Name*** gegeben.

- ***Parameter*** Liste aller Parameter, jeweils durch ein Komma getrennt, die an die Funktion übergeben werden sollen. Es gibt hierbei Funktionen, wie z.B. die **GetDesktopWindow** die keinen Parameter erwarten. Im wesentlichen gelten hier die selben Bestimmungen wie bei jeder anderen VB-Funktion auch. Das beinhaltet die Nennung des Parametertyps als auch die Schlüsselwörter **ByVal** und **ByRef**.
- ***Rückgabetyt*** Datentyp des von der Funktion erwarteten Rückgabewertes. Dieser ist bis auf wenige Ausnahmen vom Typ **Long**. Eine Ausnahme bilden hier die Subs, da sie keinen Rückgabewert besitzen entfällt auch dessen Nennung. Die Syntax einer solchen Sub sähe dann wie folgt aus:

```
[{Public | Private}] Declare Sub Funktions_Name Lib "DLL_Dateiname" [Alias  
"Funktions_Alias"] (Parameter)
```

Parameter

Diese geben an, wieviele Variablen übergeben werden und von welcher Art sie beschaffen sein müssen und regeln damit den genaueren Kontext über die Abarbeitung der Funktion. Sollte die in der Deklaration angegebene Anzahl von die Vorgabe aus der DLL nicht exakt treffen, wird bei Aufruf das Programm mit einem entsprechenden Hinweis unterbrochen. Im Laufe der Zeit werden Sie vielleicht die ein oder andere Funktion bemerken, die auch optionale Parameter zuläßt. Dies ist aber eher recht selten, grundsätzlich sollten Sie daher alle zur Funktion gehörenden Parameter, egal ob Pflicht oder optional strikt deklarieren.

Wie bereits weiter oben erwähnt entspricht der sonstige Aufbau einer gewöhnlichen VB-Funktion oder Sub. Die folgenden Zeilen sollen die zu verwendende Syntax näher erläutern:

```
[{ByVal | ByRef}] Argument As Datentyp, ...
```

- **Argument** Ein beliebiger Variablenname. Aber auch hier wieder der Hinweis möglichst Namen zu verwenden, die von den einschlägigen API-Dokumentationen bereits angeboten werden. Funktionell hat dies zwar keinen Einfluß, verbessert aber wieder eine spätere Lesbarkeit.
- **Datentyp** Der jeweils erwartete Datentyp. Dies können bekannte Variablen wie `Long`, `Byte`, `String` oder aber auch `Any` bzw. benutzerdefinierte Typen sein. Die Verwendung des Schlüsselwortes `Any` gestattet es, jeden beliebigen Variablen-Typen zu übergeben. Das ist aber mit Vorsicht zu genießen, da hier die VB-eigenen Sicherheitsmechanismen außer Kraft gesetzt werden. Sollten Sie in Verlegenheit kommen verschiedene Datentypen an ein und die selbe Deklaration übergeben zu können, ist es ratsam für jeden der Typen besser eine einzelne eigene Deklaration unter Verwendung des `Alias` Attributes anzulegen.

Unser erstes Beispiel zeigt wie praktisch eine API-Funktion deklariert und aufgerufen wird. Die verwendete Funktion `GetDesktopWindow` ist sehr leicht zu handhaben, da sie keine Parameter benötigt und nur über einen Rückgabewert verfügt:

Ermittlung des Desktop Handles [Beispiel 1]

```
Option Explicit
```

```
Private Declare Function GetDesktopWindow Lib "user32" _
    () As Long
```

```
Private Sub Command1_Click()
    Dim Desktop_hWnd As Long
```

```
    Desktop_hWnd = GetDesktopWindow
```

```
    MsgBox ("Das Fensterhandle [hWnd] des " & _
        "Desktops lautet: " & Desktop_hWnd)
```

```
End Sub
```

Verschiedene Datentypen

Wie bereits bekannt, spezifizieren Datentypen, den benötigten Speicherplatz und das Format einer Variablen. API-Funktionen unterstützen nur eine überschaubare Menge der in VB bekannten Datentypen. Da das API auf der C-Syntax basiert werden oftmals Typen benötigt, die so in VB nicht vorhanden sind.

Hier stellt sich, nebst der Voraussetzung von Pointern, die VB offiziell ebensowenig kennt, auch das grundlegende Problem im Umgang mit dem API: Die Umwandlung der in einer Deklaration erwarteten Variablen von C in VB-Syntax und umgekehrt.

Folgende Tabelle bietet eine Gegenüberstellung der gängigen Variablen Typen von C und VB

C-Typ	VB-Typ	Byte	Bereich
char	-	1	-128 bis 127
unsigned char	Byte	1	0-255
short	Integer	2	-32768-32767
unsigned short	-	2	0 bis 65535
long	Long	4	-2147483 648 bis 2147483647
unsigned long	-	4	0 bis 4294967295
int	Long	4	-2147483 648 bis 2147483647
unsigned int	-	4	0 bis 4294967295
struct	UDT	variabel	Je nach Definition
float	Single	4	+/-1.4E-45 bis +/-3.4E38
double	Double	8	+/-4.9E-324 bis +/- 1.7E308
ULARGE_INTEGER	Currency	8	0 bis &HFFFFFFFFFFFFFFFF
-	Boolean	2	0, -1
-	String	variabel	Je nach Definition
-	Object	4	-
-	Variant	16	Je nach Definition

Die gängigsten von VB aus angewandten Typen sind in der Reihenfolge: Long, UDT, String und Byte. Unsere derzeitigen, auch noch vorzeichenbehafteten, 16-Bit Integers fallen, seitdem Windows ein 32-Bit-System ist, vollkommen heraus. Der Begriff UDT verlangt eine kleine Übersetzung und bedeutet *User Defined Types*, also benutzerdefinierte Typen, was in C den Strukturen entspricht.

Auffallend ist, daß C anscheinend keinen expliziten Datentyp für String und Boolean zu bieten hat. Tatsächlich werden diese aber in einer C-Anwendung durch Bildung eines eigenen Datentypen generiert. Dazu später mehr.

Beachten Sie bitte auch den in der Tabelle kursiv ausgerichteten *ULARGE_INTEGER*-Typen, der dazu dient vorzeichenlose 64-Bit Werte aufzunehmen. Ich bin mir nicht ganz sicher ob er in C mittlerweile tatsächlich existiert, in VB gibt's ihn zumindest explizit so nicht. Beholfen werden kann sich hier zum einen durch die Anlegung eines Typen mit zwei aufeinanderfolgenden *Long*-Variablen oder dem *Currency*-Typen. Ersteres ist auch das mir bis dato bekannte Verfahren unter C, es ist aber durch aus möglich, daß dort nachgezogen wurde und deshalb eine "normale" 64-Bit Variable zur Verfügung steht. Wie gesagt unter VB können wir uns mit einem Typen helfen oder aber wahlweise ein paar krumme Dinger mit dem Datentyp *Currency* drehen, da er vom Aufbau her über die notwendigen 8 Byte verfügt. Dazu werden wir später im Kapitel "Strukturen und Typen" noch näheres erfahren.

Was sind Pointer?

Der Begriff Pointer, übersetzt Zeiger, kommt aus dem C-Milieu. Ihrer Erklärung ist recht einfach. Variablen besitzen neben ihrem eigentlich wichtigen Wert auch eine Speicheradresse an der dieser Wert abgelegt wird und exakt das ist der Pointer. Ein Zeiger ist also nichts anderes als die Speicheradresse einer Variablen. Da wir uns im 32-Bit Zeitalter befinden, ist ein Zeiger, auf Grund des mit 32-Bit ansprechbaren Adressraumes, auch 4 Bytes gleich 32-Bit lang. Jede Variable besitzt einen solchen. Leider ist es in VB standardmäßig nicht vorgesehen mit Zeigern zu operieren. Dieser Umstand stellt sich bei Nutzung des Windows-APIs manchmal als äußerst störend heraus, da Pointer des öfteren von ihm bei der Funktionsübergabe erwartet werden. Solange eine Variable mit `ByRef` direkt referenziert wird, stellt dies kein weiteres Problem dar, da VB hier intern zwangsläufig mit der Speicheradresse arbeiten muß. Anders schaut es jedoch aus wenn ein Pointer als Wert abverlangt wird.

Eine Lösung ist aber auch hier greifbar. VB verfügt über einige, wenig dokumentierte Funktionen, die die direkte Ermittlung des Zeigers einer Variablen gestatten. Leider schweigt sich die Hilfe zu diesen sehr nützlichen Befehlen mehr als aus. Zu ihnen gehören:

- `VarPtr` ermittelt den Pointer einer numerischen Variablen.
- `StrPtr` ermittelt den Pointer eines Strings.
- `ObjPtr` ermittelt den Pointer eines Objekts.
- `VarPtrArray` ermittelt den Pointer eines Feldes.

Die Anwendung der geheimen VB-Befehle hingegen gestaltet sich, wie das untenstehende Beispiel schnell erkennen läßt relativ einfach.

Da `VarPtrArray` nicht wie die anderen drei standardmäßig bereits implementiert ist, muß eigenes eine eigene Deklaration, die lustigerweise auf die VB-Laufzeitbibliotheken, zugreift definiert werden. Bitte ändern Sie deshalb vor dem ersten Start, entsprechend Ihrer verwendeten VB-Version, die Deklaration der API-Funktion `VarPtrArray`, sie lautet für VB6 einen Hauch anders.

Rückgabewert der vier versteckten Funktionen ist immer ein 32-Bit Longwert. Sollte der auf diese Art ermittelte Zeiger in eine Long-Variablen zwischengespeichert werden, ist bei dessen Verwendung im Aufruf einer API-Funktion unbedingt zu beachten, daß die Variable mit dem Schlüsselwort `ByVal` übergeben wird. Andernfalls wird nicht der gewünschte Wert, sondern der Zeiger auf die Variable in der sich der Wert befindet übermittelt, und das ist ein gravierender, manchmal auch folgenschwerer Unterschied! Wir werden im nächsten Kapitel genaueres über den feinen und gar nicht so kleinen Unterschied zwischen `ByVal` und `ByRef` erfahren.

Pointer von Variablen ermitteln [Beispiel 2]

```
Option Explicit
```

```
'Version für VB5
```

```
Private Declare Function VarPtrArray Lib "msvbvm50.dll" Alias _
    "VarPtr" (Ptr() As Any) As Long
```

```
'Version für VB6
```

```
'Private Declare Function VarPtrArray Lib "msvbvm60.dll" Alias _
'    "VarPtr" (Ptr() As Any) As Long
```

```
Private Sub Command1_Click()
```

```
    Dim Pointer As Long
```

```
    Dim Zahl As Long
```

```
    Dim Feld(0 To 1) As Long
```

```
    Dim Zeichen As String
```

```
        'Pointerermittlung von numerischen Variablen
```

```

Zahl = 100
Pointer = VarPtr(Zahl)
MsgBox ("Pointer auf die Long-Variablen: " & Pointer)

'Pointerermittlung von Feldern
Feld(0) = 200
Feld(1) = 300
Pointer = VarPtrArray(Feld)
MsgBox ("Pointer auf das Arrays: " & Pointer)

'Pointerermittlung von Strings
Zeichen = "HalliHallo"
Pointer = StrPtr(Zeichen)
MsgBox ("Pointer auf den Strings: " & Pointer)

'Pointerermittlung von Objekten
Pointer = ObjPtr(Command1)
MsgBox ("Pointer auf Command1: " & Pointer)
End Sub

```

ByVal und sein Kumpel ByVal

Die Bedeutung der beiden Schlüsselwörter ist im Zusammenhang mit dem API äußerst wichtig, von daher ist diesem Kapitel bitte eine erhöhte Aufmerksamkeit schenken.

- **ByVal** Steht für "By Value", was soviel bedeutet wie "als Wert". Dies erklärt auch gleich seine Funktion: Eine als mit ByVal übergebene Variable wird als Wert übergeben, d.h. es wird eine Kopie erstellt und nur dieser an den Prozedurkopf einer Funktion oder Sub weitergegeben. Wird in Folge, in der Funktion, der Wert geändert, hat das keine rückwirkenden Auswirkungen auf die ursprünglich übergebene Variable. Der Zusammenhang wird vielleicht durch die Erläuterungen zum Schlüsselwort **ByRef** deutlicher.
- **ByRef** Steht für "By Reference". Variablen die auf diese Weise übergeben wurden, sind dadurch direkt referenziert. Das bedeutet es wird nicht ein Wert übergeben, sondern lediglich ein Zeiger auf die Position der Variablen im Speicher. Das hat zur Folge daß im Gegensatz zu **ByVal** alle Änderungen dieser Variablen in der bearbeitenden Funktion oder Sub direkt auf die ursprüngliche Variable einwirken, da ja von hier aus direkt darauf verwiesen wird. **ByRef** ist unter VB standardmäßig eingestellt, d.h. wird das Schlüsselwort **ByVal** nicht explizit angegeben, gilt automatisch **ByRef**.

Folgendes Beispiel verdeutlicht den Unterschied von **ByVal** und **ByRef** ohne Verwendung einer API-Funktion recht gut:

ByVal & ByRef [Beispiel 3]
Option Explicit

```

Private Sub Command1_Click()
    Const Original As Long = 100
    Dim Variable As Long

    Variable = Original
    Call ByVal_Demo(Variable)
    MsgBox ("ByVal Demo: Die Variable hatte vor " & _
           "dem Aufruf den Wert:" & Original & _

```

```
    " und danach ebenfalls den Wert: " & _  
    & Variable)  
  
    Variable = Original  
    Call ByRef_Demo(Variable)  
    MsgBox ("ByRef Demo: Die Variable hatte vor dem " & _  
        "Aufruf den Wert:" & Original & _  
        ", danach allerdings den Wert: " & _  
        Variable)  
End Sub
```

```
Private Sub ByVal_Demo(ByVal Parameter As Long)  
    Parameter = Parameter + 1  
End Sub
```

```
Private Sub ByRef_Demo(ByRef Parameter As Long)  
    Parameter = Parameter + 1  
End Sub
```

Wird eine Variable also als **ByVal** übergeben, erstellt VB einen Wert und übergibt diesen an die Funktion. Genaugenommen, legt sie den Wert in einem Prozedurstack ab, der auf der anderen Seite von der aufgerufenen Funktionen wieder herausgeholt wird. Spielen wir zum besseren Verständnis folgendes Szenario durch:

Szenario 1: Eine API-Funktion erwartet als Parameter einen Zeiger auf eine Long-Variable. Diese besitzt den Wert &H100 und hat ihren Wohnsitz in der Speicheradresse &H200000. Übergeben wir die Variable vorerst mit dem Schlüsselwort **ByRef**, was passiert? Recht einfach, die Speicheradresse &H200000 wird als Wert auf den Stack gelegt, die Funktion ruft diese ab und kann somit direkt auf den Variableninhalt verweisen [sie erwartet es ja auch nicht anders].

Szenario 2: Jetzt übergeben wir unsere Variable mit **ByVal**. Was aber passiert nun? Es wird hinter den Kulissen eine Kopie erstellt, und der Wert &H100 wandert auf den Stack, die Funktion wiederum versucht nun diesen Wert aus dem Stack zu holen was auch noch gelingt. Für sie stellt sich aber &H100 jetzt nicht wie angedacht als Wert dar, sondern, wie bei Szenario 1, als Speicheradresse. Bei dem Versuch diese zu referenzieren, gelangt sie daher aller Wahrscheinlichkeit nach in einen Speicherbereich der ihr gar nicht zusteht, die Folge ist eine allgemeine Schutzverletzung und ein abrupter Programmabbruch.

Aus diesem Grund ist die Verwendung der beiden Schlüsselwörter peinlichst genau abzuwägen und den jeweiligen Anforderungen der Funktion anzupassen. Als Faustformel läßt sich sagen, daß **ByVal** angewandt werden darf, wenn davon auszugehen ist, daß die aufzurufende Funktion keine Änderungen an der ursprünglichen Variable vornimmt. Die beschriebene Methodik trifft auf alle VB-Variablen zu bis auf eine Ausnahme, die Zeichenfolgen [Strings], daher ist diesen Unholden weiter unten auch ein eigenes Kapitel gewidmet.

Zeigerspiele

Nun, da Sie jetzt in etwa wissen was Zeiger, respektive Pointer, sind und Sie genaueres über die Funktionsweise von `ByVal` und `ByRef` erfahren haben, sollten Sie Ihre Kenntnisse an ein paar praktischen Beispielen festigen.

Rekapitulieren wir und fassen kurz zusammen: Zeiger sind Verweise auf die physikalische Speicheradresse einer Variablen. Übergeben wir an eine Funktion eine Variable mit `ByRef` wird deren Zeiger übergeben, mit `ByVal` ihr Wert.

Was jetzt noch fehlt ist ein Werkzeug, daß es uns gestattet Speicheroperationen mit vorhandenen Zeigern durchzuführen. Unter VB selbst sind wir, einmal vor dieses Problem gestellt, auf weiter Flur mehr als nur verlassen. Glücklicherweise bietet uns das API hier eine Krücke in Form von diversen Speicheroperation-Funktionen, namentlich benannt durch: `CopyMemory`, `FillMemory`, `MoveMemory` und `ZeroMemory`. Diese vier Grazien im Kampf gegen den Pointerismus kommen uns wie gerufen. Im wesentlichen ist uns die Funktion `CopyMemory` wichtig, die restlichen sind lediglich Spielarten der ersten. Schauen wir uns deren Deklaration daher einmal an:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
    (lpTo As Any, lpFrom As Any, ByVal lLen As Long)
```

`CopyMemory` kopiert wie der Name schon vermuten läßt einen Speicherblock in den eines anderen. Die beiden Speicheradressen werden durch Pointer beschrieben. Wenn sich Quell- und Zielsegment überschneiden ist das Resultat unsicher, für einen solchen Vorgang ist besser `MoveMemory` zu nehmen. Mit `CopyMemory` kann durch Verwendung des Schlüsselwortes `ByVal` aus einem Variablenwert ein Zeiger, vorzugsweise `Long`, übergeben werden. Dies ist z.B. interessant um von API-Funktionen erhaltene Zeiger in eine VB-Variable zu kopieren, danach zu ändern und abschließend wieder zurückzukopieren. Für den letzteren Vorgang ist es notwendig eben einen solchen obengenannten Zeiger zu besitzen bzw. zu generieren. [siehe auch `CopyMemoryPtr`]

Aha, daß hört sich doch sehr gut an. Es ist also hiermit möglich bei lediglichem Vorhandensein eines Zeigers, diesen an einen, wiederum durch einen Zeiger bestimmten, Speicherplatz zu kopieren. Die Position des Zielsegments könnte also auch durch den Pointer einer VB-eigenen Variable beschrieben werden. Das versetzt uns jetzt in die phänomenale Lage eine x-beliebige, nur durch einen Zeiger beschriebene Variable in eine normal handhabbare VB-Variable zu konvertieren.

Wichtige Bedingung bei diesem Kopiervorgang ist zum einen das tatsächliche Vorhandensein des Ziels und zum anderen, daß dort ausreichend Speicherplatz zur Verfügung steht. Ist letzteres nicht gegeben, werden beim Kopieren bestenfalls andere Daten überschrieben oder schlimmstenfalls die Anwendung mal wieder wegen eines unberechtigten Speicherzugriffs abrupt geschlossen. Um den Vorgang sauber vonstatten gehen zu lassen, sollte daher, wenn die Quelle beispielsweise einen `Long`-Wert als Pointer anbietet, im Ziel auch ein `Long`-Wert vorliegen.

Dies wird deutlicher im untenstehenden Beispiel. Dort wird zu aller erst einer `Long`-Variablen ein beliebiger Wert zugewiesen, um danach mit Hilfe der undokumentierten VB-Funktion `VarPtr` den ermittelten Zeiger in der Variablen `Pointer` zwischenspeichern. Bis hier hin ist alles nur Simulation, die der Verstehbarkeit des Beispiels dienlich sein soll. Tatsächlich könnten wir uns das Zuweisen und speziell das Pointerermitteln sparen, da wir letzteren ja von ein API-Funktion frei Haus geliefert bekämen.

Jetzt wird's spannend. Mit der Zeile `Ziel = PointerToLong(Pointer)` rufen wir die Funktion `PointerToLong` auf die den Wert hinter unserem Pointer ermitteln soll. Zu diesem Zweck müssen wir die Variable `Pointer` an besagte Funktion übergeben und das als `ByVal`! Warum das? Richtig, da sonst der Zeiger auf die Variable `Pointer` übermittelt würde und nicht wie gewünscht nur der Wert der Variablen.

In der Funktion passiert weiterhin nicht viel, kommen wir also zur Kernzeile:

```
Call CopyMemory(Buffer, ByVal (Ptr), 4)
```

Aufgeschlüsselt bedeutet dies, daß ab der durch `Ptr` referenzierten Speicherzelle, 4 Bytes in die durch die Variable `Buffer` referenzierte Speicherzelle kopiert werden. Da der Variablentyp `Long` exakt 4 Byte lang ist, liegt nach Abschluß des Kopiervorgangs, der hinter dem ursprünglichen Pointer verborgene Wert, als normale, in VB nutzbare Variable in `Buffer` vor. Besonders hervorzuheben ist auch hier wieder die Verwendung des Schlüsselworts `ByVal` um zu verhindern, daß aus den Gleichen Gründen wie bei der Funktionsübergabe die Variable `Ptr` direkt referenziert wird.

Pointer in eine Variable kopieren [Beispiel 4]

```
Option Explicit
```

```
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
    (lpTo As Any, lpFrom As Any, ByVal lLen As Long)
```

```
Private Sub Command1_Click()
    Dim Quelle As Long, Ziel As Long
    Dim Pointer As Long

    'x-belieigen Wert zuweisen
    Quelle = 4711

    'Zeiger generieren
    Pointer = VarPtr(Quelle)

    'Wert aus Zeiger auslesen
    Ziel = PointerToLong(Pointer)

    'Ergebnis Ausgeben
    MsgBox ("Die Variable 'Quelle' mit dem Pointer '&H' & _
        Hex(Pointer) & "' hat den Wert '" & Quelle & _
        "'.'" & vbCrLf & "Nur durch das bloße " & _
        "Vorhandensein dieses Zeigers ist es bereits " & _
        "möglich an ihren Wert zu gelangen." & vbCrLf & _
        "Die referenzierte Variable wurd in 'Ziel' " & _
        "abgelegt und besitzt wie sein Original den" & _
        " Wert '" & Ziel & "'")
End Sub
```

'Die eigentliche Umwandlung

```
Private Function PointerToLong(Ptr As Long) As Long
    Dim Buffer As Long

    'Kopiert vom angegeben Pointer "Ptr" 4 Zeichen in den
    'durch "Buffer" referenzierten Speicherbereich.
    '4 Byte deshalb, da ein Longwerte genau diesen Platz
    'einnimmt
    Call CopyMemory(Buffer, ByVal (Ptr), 4)

    PointerToLong = Buffer
End Function
```


Boolean und Arrays

Zu beachten ist speziell beim Typ `Boolean` [C-Äquivalent `-> bool`], daß der Wertebereich bei 0 für `False` und 1 für `True` liegt. In Visual Basic hingegen ist `False` zwar auch der Wert 0 zugeordnet, `True` jedoch entspricht `-1`. Dies ist sowohl bei der Auswertung des Rückgabewerts einer API-Funktion, als auch beim Setzen eines Flags vor dem Aufruf, ein oft gemachter Fehler.

Eventuell hat sich Ihnen schon die Frage aufgedrängt, wie denn Arrays, sprich Felder, an Funktionen des APIs übermittelt werden können. Die Lösung ist unerwartet leicht. In der Deklaration selbst bleibt die Verwendung eines Feldes unberücksichtigt. Es wird lediglich eine Variable ohne Dimension des entsprechenden Arraytyps deklariert. Übergeben wird bei Aufruf lediglich das erste Element des Arrays. Die Funktion verfügt dann auf jeden Fall über einen weiteren Parameter, in der die Obergrenze des Feldes abgelegt wird. Somit ist der API-Funktion nach Aufruf, sowohl das erste als auch das letzte Element eines Array stets bekannt.

Es ist hier darauf zu achten, daß das Feld vor der Übergabe auf jeden Fall dimensioniert worden ist, außerdem sollte die in dem zweiten Parameter angegebene Array-Größe nie über die eigentlichen Feldgrenzen hinausragen, da es sonst seitens des APIs zu unberechtigten Speicherzugriffen kommt und Ihre Anwendung ungewollt geschlossen wird.

Arrays bzw. Felder an das API übergeben [Beispiel 5]

`Option Explicit`

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

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

```
Private Sub Timer1_Timer()
    Dim z As Integer, x As Integer

    'Das Array
    Dim Pt() As POINTAPI

    With Picture1
        .Picture = LoadPicture()

        z = Rnd * 20 + 10

        'Arrayobergrenzen definieren
        ReDim Pt(0 To z)

        'Array füllen
        For x = 0 To z
            Pt(x).x = .ScaleWidth * Rnd
            Pt(x).y = .ScaleHeight * Rnd
        Next x

        'Das Array und seine Obergrenze übergeben
        Call Polygon(.hdc, Pt(0), z)
    End With
End Sub
```

```
Private Sub Form_Load()  
    'Vorbereitungen  
    With Picture1  
        .Picture = LoadPicture()  
        .BackColor = &HFFFFFF  
        .ScaleMode = vbPixels  
        .FillColor = &HFF&  
        .FillStyle = vbFSSolid  
        .Refresh  
    End With  
  
    Timer1.Interval = 200  
    Timer1.Enabled = True  
End Sub
```

Strukturen respektive Typen und 64-Bit Integers

Beides ist das gleiche. "Typen" werden in Visual Basic verwendet und "Strukturen" sind die zugehörigen C Pendanten. Eine Struktur ist ein Variablenobjekt, das die verschiedensten Variablen unter einer einzigen zusammenfassen kann. Die einzelnen Mitglieder eines solchen Objekts werden im C-Jargon auch "Members" genannt. Ein Member kann von einem beliebigen Datentyp sein und zum Beispiel auch wieder Strukturen beinhalten.

So gesehen wird mit der Anlegung einer Struktur ein neuer Variablentyp geschaffen. Variablen können nach einer solchen Definition mit diesem neuen Typen dimensioniert werden und übernehmen somit auch dessen Struktur [daher wohl auch der Name]

Um eine Struktur unter Visual Basic zu definieren, wird der `Type` Operator verwendet. Er umschließt alle Mitglieder einer Struktur und kann wie eine Funktion, je nach benötigtem Geltungsbereich, `Private` oder `Public` sein:

```
[(Public | Private)] Type type_name  
    member1 As data_type1  
    member2 As data_type2  
    ...  
End Type
```

- **type_name** bezeichnet den Namen der Struktur. Unter diesem können dann später andere Variablen dimensioniert werden. Vereinbarungsgemäß sollte dieser Name groß geschrieben werden, dies ist aber nicht zwingend notwendig, sondern dient lediglich der Übersichtlichkeit.
- **member1, member2, ...** Die Namen der einzelnen Strukturmitglieder. Nach Zuweisung werden die Werte dieser Variablen tatsächlich also Zeiger oder direkt in der Struktur abgelegt. Es gelten die selben Konventionen wie bei herkömmlichen Variablen.
- **data_type1, data_type2, ...** Der Datentyp des einzelnen Strukturmitglieds. Gültig sind alle Datentypen, angefangen mit Byte über String bis hin zum Objekt. Strings dürfen flexibler, genauso wie auch fester Länge sein. Weiterhin ist es erlaubt eine Strukturvariable wiederum mit einer Struktur zu dimensionieren, so daß sich Verschachtelungen ergeben können. Nicht möglich ist die Verwendung der von C her bekannten, rekursiven Strukturen. Dies ist eine Besonderheit, die die Dimensionierung eines Members durch die Struktur in der das Member selbst sitzt vornimmt. Dies Form wird aber vom

API meines Wissens nach, auch nirgends erwartet, so daß das nicht Vorhandensein einer solchen Möglichkeit unter VB zu verschmerzen ist. Ansonsten gelten auch hier wieder die selben Konventionen, wie bei herkömmlichen Variablen.

Hier ein Beispiel für zwei Strukturen, wobei die zweite, die erste mit einschließt:

```
Option Explicit

Private Type RECT
    left As Long
    top As Long
    right As Long
    bottom As Long
End Type

Private Type MYTYPE
    x As Byte
    y As Byte
    R As RECT
    Text As String
End Type

Private Sub Command1_Click()
    Dim M As MYTYPE

    M.R.bottom = 0
    M.R.left = 0

    M.x = 100
    M.y = 200

    M.Text = "Test"
End Sub
```

Viele API-Funktionen erwarten die Übergabe solcher Strukturen; haben sie doch den Vorteil, größere Mengen von Daten zu umfassen, bzw. logisch zu gruppieren, und dadurch diese, durch Angabe nur eines Parameters übergebar zu machen.

Das Übermitteln von Strukturen an API-Funktionen erfolgt in der Regel denkbar einfach. Meist wird die von der Funktion erwartete Struktur bereits in der Parameterliste der Deklaration vorausgesetzt. So kann eine Struktur wie jede andere Variable übergeben werden. Da hier nie die gesamte Struktur als Wert übergeben wird, sondern wie so oft lediglich ein Zeiger, darf das Schlüsselwort `ByVal` nicht verwendet werden. Strukturen sind also grundsätzlich `ByRef` zu übergeben!

In seltenen Fällen können auch Strukturen von API –Funktionen an Visual Basic zurückgegeben werden. Allerdings nie als Wert sondern, wie nicht anders zu erwarten war, immer nur als Pointer. Speziell beim `SubClassing` wird man hin und wieder vor das Problem gestellt, eine über einen Pointer referenzierte Struktur ändern zu müssen. Aber auch hierfür gibt es eine Lösung, wieder die `CopyMemory`. Die hier angewandte Technik basiert im wesentlichen auf der im obigen Kapitel `Zeigerspiele` beschriebenen Methodik. Wir müssen sie lediglich ein bißchen erweitern und anpassen.

Das Prinzip ist leicht zu verstehen. In VB wird die zu erwartende Struktur gleichen Typs und Aufbaus angelegt und eine Variable damit dimensioniert. Anhand des von der API-Funktion zurückgegebenen Pointers, kann daraufhin mit der `CopyMemory` der durch den Pointer referenzierte Speicherblock einfach in die VB-Variable hineinkopiert werden. Im Anschluß daran sind die nötigen Änderungen vorzunehmen um zum Abschluß das ganze wieder in den originalen Speicherblock zurückzukopieren.

Einzigste Schwierigkeit ist die Ermittlung der Länge des zu kopierenden Speichersegments. Da die Struktur aber in VB bereits definiert ist, kann mit sie mit Hilfe des `Len()` Operators, der eigentlich mehr in Bezug auf Strings bekannt ist, problemlos ermittelt werden. Hier das oben besprochene Vorgehen als Beispiel:

Strukturen über Pointer handeln [Beispiel 6]

`Option Explicit`

```
Private Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
    (lpTo As Any, lpFrom As Any, ByVal lLen As Long)
```

```
Private Type RECT
    left As Long
    top As Long
    right As Long
    bottom As Long
End Type
```

```
Private Quelle As RECT
```

```
Private Sub Command1_Click()
    Dim Pointer As Long
```

```
    '-----
    'Der Einfachheit halber wird in der folgenden Abschnitt
    'eine Struktur angelegt und deren Pointer ermittelt.
    'Diese würde bei einer reinen Zeigerrückgabe durch eine
    'API-Funktion natürlich entfallen.
```

```
    Quelle.left = 10
    Quelle.top = 20
    Quelle.right = 300
    Quelle.bottom = 400
    Pointer = VarPtr(Quelle)
```

```
    '-----
```

```
    'Struktur vor der Änderung anzeigen
    Call ShowStructure
```

```
    'Struktur auslesen, ändern und zurückschreiben
    Call ManipulateStructureFromPointer(Pointer)
```

```
    'Struktur nach der Änderung anzeigen
    Call ShowStructure
```

```
End Sub
```

```
'Die eigentliche Routine
```

```
Private Sub ManipulateStructureFromPointer(ByVal Ptr As Long)
```

```
    Dim Temp As RECT
    Dim Länge As Long
```

```
    'Länge der Struktur ermitteln
    Länge = Len(Temp)
```

```
    'Kopieren des referenzierten Pointers in eine temporäre
    'gleichartige Struktur
    Call CopyMemory(Temp, ByVal (Ptr), Länge)
```

```

'Kopie der ursprünglichen Struktur befindet sich jetzt
'in "Temp" und kann beliebig manipuliert werden.
Temp.left = Temp.left * -200
Temp.top = Temp.top * -200
Temp.right = Temp.right * -200
Temp.bottom = Temp.bottom * -200

'Zurückkopieren der veränderten Struktur in das
'ursprüngliche Speichersegment
Call CopyMemory(ByVal (Ptr), Temp, Länge)
End Sub

```

```

'Diese Funktion dient nur zur Ausgabe der Werte die in der
'Struktur abgelegt sind
Private Sub ShowStructure()
    Dim Text As String

    Text = "Die Members der Struktur 'Quelle' haben " & _
           "jetzt folgende Werte:" & vbCrLf & vbCrLf & _
           ".left      = " & Quelle.left & vbCrLf & _
           ".top        = " & Quelle.top & vbCrLf & _
           ".right     = " & Quelle.right & vbCrLf & _
           ".bottom   = " & Quelle.bottom & vbCrLf

    MsgBox (Text)
End Sub

```

Ein weiteres Thema, was nicht ganz zum Kapitel Strukturen paßt, oder vielleicht doch, ist die Bildung von 64-Bit Ganzzahl-Variablen. Diese gibt es in VB so nicht, sind aber unter C eventuell bekannt [ich bin mir da nicht so sicher], auf jedenfalls werden sie von bestimmten, seltenen API-Funktionen als Eingabeparameter durchaus erwartet.

Die wohl einfachste Möglichkeit eine 64-Bit Variable nachzubilden ist die Organisation zweier aufeinanderfolgender 32-Bit `Long`-Variablen:

```

Type ULARGE_INTEGER
    LowPart As Long
    HighPart As Long
End Type

```

Da Werte oftmals und Typen grundsätzlich als Zeiger übergeben werden müssen, ist es der API-Funktion letztendlich egal, ob es sich bei der Übergabe um eine definitionsgerechte 64-Bit, also 8 Byte lange, Variable oder um zwei aus je vier Byte zusammengesetzte `Long`-Werte handelt. Richtig angeordnet sehen im Speicher beide identisch aus. So ließe sich auf die selbe Art und Weise auch problemlos ein 96-Bit oder gar 128-Bit etc. Variablentyp entwickeln.

Ein eine weitere Methode an 64-Bit Variablen zu gelangen ist, die Verwendung des Typs `Currency`. Er entspricht mit seinen 8-Byte Länge exakt unseren Anforderungen. Zur Zwischenpufferung eines 64 Bit Wertes ist er wie geschaffen. Für alles weitere, wie z.B. Rechenoperationen, ist er nur bedingt einsatzfähig. VB hat nämlich die unangenehme Eigenart den in einer `Currency` Variablen enthaltenen Wert durch Plazierung eines Dezimalpunktes um den Faktor 10.000 herabzusetzen, damit die erforderliche Genauigkeit in den Nachkommastellen erzielt werden kann.

Wollen wir diesen Typen nach einer erfolgten Rechenoperationen anzeigen, ist dies zu berücksichtigen und der Wert entsprechend mit 10.000 aufzumultiplizieren, um somit die unerwünschte Kommastelle zu eliminieren. Dies hat aber wiederum aber eine Reduktion der visuell darstellbaren Gesamtstellen zur Folge, in den meisten Fällen sollte das jedoch ausreichen: Hier ein Beispiel:

64 Bit Integer vom Typ Currency [Beispiel 7]

```
Option Explicit
```

```
Private Declare Sub CopyMemory Lib "kernel32.dll" Alias _
    "RtlMoveMemory" (Dest As Any, Src As Any, ByVal _
    Length As Long)
```

```
Private Type ULARGE_INTEGER
    LowPart As Long
    HighPart As Long
End Type
```

```
Private Sub Command1_Click()
    Dim I64_t As ULARGE_INTEGER
    Dim I64_c As Currency
    Dim Temp As Currency

    'Anlegen einer 64 Bit Zahl über die
    'konventionelle Methode einer Struktur
    I64_t.HighPart = 23
    I64_t.LowPart = 1215752192

    'Kopieren der Struktur in den Currencytypen
    Call CopyMemory(I64_c, I64_t, 8)

    'Löschung des Kommas
    Temp = I64_c * 10000

    'Ausgeben des 64-Bit Wertes
    MsgBox ("Die Variable 'I64_c' ist vom Typ Currency " & _
        "und besitzt derzeit den 64-Bit Wert '" & _
        & Format(Temp, "###,###,###,###") & "'")

    'Rechnen mit dem 64-Bit Wert
    I64_c = I64_c * 2 + (1 / 10000)

    'Löschung des Kommas
    Temp = I64_c * 10000

    MsgBox ("Nach Anwendung einer Multiplikation und einer" & _
        " Addition, " & vbCrLf & "besitzt Variable " & _
        "'I64_c' vom Typ Currency nun den " & vbCrLf & _
        "64-Bit Wert '" & Format(Temp, "###,###,###,###") & _
        & "'")
End Sub
```

Die Zeichenfolge als Querulant

Strings nehmen eine besondere Position bei der Übergabe an eine API-Funktion ein. Lassen Sie uns ersteinmal eine grundsätzliche Überlegung zum Aufbau eines Strings tätigen.

Ein String ist eine Zeichenfolge, deren erstes Zeichen an einer bestimmten Stelle im Speicher steht. Jedes weitere Zeichen folgt schön der Reihe nach dem ersten, so daß sich eine Kette ergibt. Im einfachsten Falle könnte sich die Übergabe einer solchen Kette an eine API-Funktion wie folgt abspielen:

Mittels `ByRef` wird ein Zeiger auf den Anfang des nur in unserer Vorstellung existierenden Strings übergeben, so daß die Funktion die Position des ersten Zeichens im Speicher kennt. Das sähe dann so aus:



Nun verhält es sich aber so, daß in VB sämtliche Strings im Unicodeformat gehandhabt werden, sprich für jedes Zeichen werden grundsätzlich zwei Bytes reserviert. Zudem handelt VB Zeichenfolgen in einem eigenen Format, dem sogenannten BSTR [Basic String]. Das hat zur Folge, daß der eigentlichen Zeichenfolge ein vier Byte langer Deskriptor vorangestellt ist. Somit kommen wir zu folgendem Bild:



Der besagte Deskriptor bezeichnet lediglich die Länge der Zeichenfolge, da hier vier Bytes zur Verfügung stehen, liegt nahe, daß Strings im BSTR-Format maximal eine Länge von 4.294.967.296 Zeichen haben können, da das ausreicht sind wir sehr zufrieden und schauen uns das ganze jetzt in Form eines Speicherauszugs an. [die vorher lesbaren Zeichen sind jetzt Werte]



Wir sehen also im Deskriptor, daß die im Unicode vorliegende Zeichenkette insgesamt 18 Bytes umfaßt, damit ist das Ende des Strings eindeutig festgelegt.

Soweit, so gut, das war jetzt BASIC. Dummerweise kennt die API keine BSTR-Strings, so daß auf Grund der dort vorherrschenden C-Syntax ein anderes Stringformat anzuwenden ist. Hier werden nämlich nullterminierte Zeichenfolgen ohne vorangestellten BSTR-Deskriptor erwartet. Einen solchen erhält man zum Beispiel durch einfaches Anhängen mit `Chr$(0)` oder mit der vordefinierten VB-Konstante `vbNullChar`. Daraus folgt, daß für die uns beiden bekannten Varianten [ANSI und Unicode] ein String auf die unten dargestellten Weisen von einer API-Funktion erwartet werden.



Das wäre dann geklärt, aber wie bringen wir die eine Form in die andere und umgekehrt und vor allem, was machen wir mit dem jedem String anhaftenden Deskriptor? Wir haben Glück, denn VB schneidet ihn bei der Übergabe an eine API-Funktion automatisch ab. Und VB macht darüber hinaus noch mehr, es wandelt nämlich ungefragt sämtliche in Unicode vorliegende Zeichenfolgen in ANSI-Code um.

Die Gründe für dies Nettigkeit kommen nicht von ungefähr und liegen darin begründet, daß wir uns, wie bereits schon einmal Eingangs erwähnt, in einer Übergangsphase von ANSI zu Unicode befinden. VB arbeitet bereits allumfassend mit diesem neuen Format. Nicht aber das API. So müssen wir uns umständlicher Weise bei API-Funktionen, welche Strings erwarten, immer mit zwei Versionen herumärgern. Nämlich zum einen mit der ANSI- und zum anderen mit der Unicode-Variante. Speziell letztere existiert unter Win9x [NT bildet hier die Ausnahme] Systemen oftmals nur als leere Hüllfunktion. Das ist Microsofts Rückversicherung für eine Zeit in der es nur noch Unicode und kein ANSI mehr geben wird.

Zu unterscheiden sind die beiden Versionen durch je einen nachgestellten Großbuchstaben. So gibt es die W-Variante für Unicode und die A-Variante für ANSI. In Hinsicht darauf, daß wegen der oben genannten Gründe nicht gewährleistet ist, daß sich hinter einer W-Variante auch tatsächlich etwas verbirgt, greifen wir grundsätzlich bei Verwendung von Stringfunktionen unter Zuhilfenahme des `Alias` Schlüsselwortes immer auf die A-Version zurück.

Zurück zum Thema. Wir wissen jetzt also, daß VB an das API zu übergebende Strings automatisch von BSTR in ANSI wandelt und den Deskriptor abschneidet. Bedingung zur Einleitung dieses Vorgangs ist hier und nur hier, entgegen aller sonstigen Gepflogenheiten, der Einsatz des Operators `ByVal`. Dies ist sehr wichtig! Erlaubt es uns doch im C-Sinne Strings, wie dort erwartet, an API-Funktionen weiterzugeben.

Intern erstellt VB im Speicher tatsächlich, wie bei der Verwendung von `ByVal` zu vermuten war, nebst den eben erwähnten Aktionen, eine Kopie der Zeichenfolge und übergibt diese der Funktion als Zeiger. Es besteht jetzt die Möglichkeit, daß die API-Funktion Änderungen an der übermittelten Kopie vornimmt, speziell z.B. Zeichen in sie hineinschreibt. Deshalb sollte der als Puffer fungierende String von VB aus entsprechend groß angelegt sein. Dies geschieht unter Verwendung der Befehle `Space$(Groesse)` oder `Puffer = String$(Groesse, Chr$(0))`.

Hat bei ausreichend großem String soweit alles geklappt, geht nach Beendigung der Funktion die Kontrolle an VB zurück. Dort wird dann der erhaltene und geänderte ANSI-String wieder in Unicode gewandelt und der im Speicher befindliche immer noch temporäre in sein Original zurückkopiert.

Diese Vorgänge sind für uns transparent, daß wir nehmen sie weder direkt wahr noch haben wir einen Einfluß darauf. Deshalb kann das folgende Beispiel auch nur den grundsätzlichen Umgang mit Strings demonstrieren.

Dazu gehört vor allem die Auswahl der A-Variante einer API-Funktion, sowie das Bereitstellen eines genügend großen Stringpuffers um den der Funktion abverlangten String aufnehmen zu können. Dabei wird der Puffer als normaler Parameter an die Funktion übergeben und nicht wie vielleicht erwarte als Rückgabewert vorangestellt. Der Rückgabewert bei API-Funktionen mit Stringcharakter dient meist dazu über Art des Erfolgs oder Mißerfolg eines Aufrufs Auskunft zu geben. So gibt es bestimmte Funktionen die diesen Wert nutzen, um anzuzeigen wieviele Zeichen in den tatsächlich geschrieben wurden. Dabei wird das Terminierungszeichen, die Null, nicht berücksichtigt.

Manchmal besteht auch die Möglichkeit durch einen ersten Aufruf der Funktion die insgesamt benötigte Puffergröße vorab abzurufen. Hierfür ist der Parameter für die Puffergröße auf 0 zu setzen. Der Puffer kann dann entsprechend angelegt und dann in einem zweiten Aufruf gefüllt werden.

Auch gibt es die Variante, bei der lediglich über den Rückgabewert ein zu kleiner Puffer bemäkelt wird. Hier ist das Problem idealerweise nur durch eine sukzessive Approximation zu lösen, sprich den Puffer einer Schleife stetigen Aufrufens der API-Funktion immer weiter aufzublasen, bis er der den Anforderungen genüge trägt.

All dies wollen wir hier nicht weiter ausführen, da dies nur Einzelfälle sind und den Rahmen sprengen würde. Es sei einfach nur erwähnt, so daß Sie sich an entsprechender Stelle an diese Hinweise erinnern können. Hier folgt nun endlich ein überschaubares Beispiel:

Umgang mit Strings beim API [Beispiel 8]

Option Explicit

```
Private Declare Function GetWindowTextLength Lib _
    "user32" Alias "GetWindowTextLengthA" (ByVal _
    hwnd As Long) As Long

Private Declare Function GetWindowText Lib "user32" Alias _
    "GetWindowTextA" (ByVal hwnd As Long, ByVal _
    lpString As String, ByVal cch As Long) As Long
```

```
Private Sub Command1_Click()
    Dim l As Long, Buffer As String, Result As Long

    'Dem Fenster einen beliebigen Text zuweisen
    Me.Caption = "Schöner Text"

    'Länge des Fenstertextes ermitteln
    l = GetWindowTextLength(Me.hwnd) + 1

    'Puffer erstellen
    Buffer = Space$(l)

    'Text holen
    Result = GetWindowText(Me.hwnd, Buffer, l)

    'Auf Erfolg prüfen
    If Result <> 0 Then
        Buffer = Left$(Buffer, Result)
    Else
        Buffer = ""
    End If

    'Ergebnis ausgeben
    MsgBox ("Der Title dieses Fesnters lautet: '" _
        & Buffer & "'")
End Sub
```

Interessant ist noch zu wissen, daß Strings flexibler Länge in Strukturen lediglich als Zeiger eingebunden werden und somit dort nur 4-Byte Platz einnehmen, sprich die Länge einer solchen Struktur vergrößert sich bei Unterbringung eines solchen String um insgesamt 4. Zur Veranschaulichung wird in dem folgenden Beispiel eine Struktur erstellt, die lediglich einen String als Member enthält. Erstaunliches liefert dabei die Abfrage nach der Größe der Struktur.

Beweis eines Stringzeigers in einer Struktur [Beispiel 9]

```
Option Explicit
```

```
Private Type STRUCT
    Zeichenkette As String
End Type
```

```
Private Sub Command1_Click()
    Dim Typ As STRUCT
    Dim l As Long

    Typ.Zeichenkette = "12345678901234567890"
    l = Len(Typ)
    MsgBox ("Erstaunlich, der UDT mit dem String '" & _
        Typ.Zeichenkette & "' ist nur " & l & _
        " Byte lang!")

    Typ.Zeichenkette = "1234567890"
    l = Len(Typ)
    MsgBox ("Ebenso erstaunlich, der UDT mit dem neuen" & _
        " String '" & Typ.Zeichenkette & _
        "' ist auch nur " & l & " Byte lang!")
End Sub
```

Der Code zeigt, die Länge der Struktur ist immer 4-Byte groß, also exakt die Größe eines 32-Bit Zeigers. Um so merkwürdiger erscheint dieses Beispiel, wenn es auf einen String fester Länge geändert wird:

Strings fester Längen in Strukturen [Beispiel 10]

```
Option Explicit
```

```
Private Type STRUCT
    Zeichenkette As String * 20
End Type
```

```
Private Sub Command1_Click()
    Dim Typ As STRUCT
    Dim l As Long

    Typ.Zeichenkette = "12345678901234567890"
    l = Len(Typ)
    MsgBox ("Erstaunlich, der UDT mit dem String fester " & _
        "Länge '" & Typ.Zeichenkette & "' ist " & _
        "entgegen einem normalen String, wie zu " & _
        "erwarten, " & l & " Byte lang!")
End Sub
```

Als Ergebnis erhalten wir die tatsächliche Länge des String. Er wird also in seiner vollen Länge in die Struktur eingebettet. Dies gilt allerdings nur in Verbindung mit Strukturen, alleinstehende Strings fester Länge müssen wie oben gelernt nach wie vor mit `ByVal` übergeben werden.

Ein weiteres Problem, im Zusammenhang mit Strukturen öfters auftretend, ist das Rekonstruieren bzw. Kopieren eines Strings aus einem in einer Struktur zurückgegebenen Pointers. Dies stellt uns zuerst einmal vor ein schier unlösbares Problem, da wir nur den Ort und noch nicht einmal die Länge für einen eventuell vorzubereitenden Puffer kennen. Aber auch hier kommt uns das API wieder zu Hilfe, diesmal in Form der Funktionen `lstrcpy` und `lstrlen`. Der Trick ist denkbar leicht. Mit `lstrlen` wird zuerst die Länge des mit dem Pointer verbundenen Strings ermittelt, um einen entsprechend großen Puffer anzulegen. In diesen wird abschließend mit `lstrcpy` der ANSI-String aus dem Speicher hineinkopiert:

Pointer in einen String wandeln [Beispiel 11]

Option Explicit

```
Private Declare Function lstrcpy Lib "kernel32.dll" _
    Alias "lstrcpyA" (ByVal lpString1 As Any, _
    ByVal lpString2 As Any) As Long
```

```
Private Declare Function lstrlen Lib "kernel32" Alias _
    "lstrlenA" (ByVal lpString As Any) As Long
```

```
Private Sub Command1_Click()
    Dim Zeichen As String
    Dim Pointer As Long

    '-----
    'Dieser Teil erstellt lediglich einen ANSI-String
    'und den zugehörigen Pointer. Normalerweise würde
    'hier ein API-Aufruf stehen der nur den Pointer zu-
    'rückgibt. Der einfacheithalber wurde hierauf ver-
    'zichtet.

    'BSTR String im Unicode Format definieren
    Zeichen = "Dieser String wurde aus einem Pointer extrahiert"

    'Unicode-String in ANSI-String wandeln
    Zeichen = StrConv(Zeichen, vbFromUnicode)

    'Pointer auf unseren Fake String auslesen
    Pointer = StrPtr(Zeichen)
    '-----
    MsgBox PointerToString(Pointer)
End Sub
```

```
'Das eigentliche Extrahieren des Strings aus dem Pointer
Private Function PointerToString(ByVal Ptr As Long) As String
    Dim Länge As String, Puffer As String

    Länge = lstrlen(Ptr)
    Puffer = Space(Länge)
    Call lstrcpy(Puffer, Ptr)

    PointerToString = Puffer
End Function
```

Handles

Handles sind eine interne Definition der Windows API. Sie werden angelegt um die ganze Vielzahl der unter Windows vorhandenen Objekte systemweit verwaltbar bzw. ansprechbar zu halten. Zu diesen Objekten gehören zum Beispiel thematische Gruppen, wie Dateien, Fenster, Registryschlüssel, Pinsel, Icons, Menüs und viele andere Dinge.

Grundsätzlich bieten Handles dem Anwendungsentwickler die einzige Möglichkeit, Einfluß auf die erwähnten Objekte zu nehmen oder mit ihnen einen Umgang zu pflegen. Hinter den Kulissen verweist ein Handle auf eine interne Liste in der die verschiedensten Informationen und Eigenschaften über dessen Besitzer gespeichert sind.

Ein Handle an sich ist ein ganz normaler 32-Bit Wert. Auf Grund dieser beabsichtigten Beschaffenheit, als Aufrufparameter für eine API-Funktionen ausgezeichnet geeignet. Unter Visual Basic wird ein Handle in einem `Long`-Datentyp gespeichert und ist daher von jeder anderen 32-Bit Zahl nicht mehr zu unterscheiden.

Bedeutung gewinnt ein Handle erst, wenn es in einem bestimmten Kontext als Parameter an eine API-Funktion übergeben wird. Wie bereits Eingangs erwähnt, haftet den meisten Objekten ein solches Handle an. Dazu gehören Formulare wie auch CommandButtons, PictureBoxen und all die anderen aus VB bekannten Objekte. Praktischerweise verfügen sie bereits von Hause aus über eine direkt abfragbare Eigenschaft in der das Handle gespeichert ist. Diese Property nennt sich `.hWnd` und ist wie gefordert vom Typ `Long`.

Das Handle eines CommandButtons kann beispielsweise über `Command1.hWnd` abgefragt werden und somit einer API-Funktion als Parameter dienen. Damit wird klar, daß im Windowssinne Handleeigner wie Formulare, CommandButtons, ListBoxen, ComboBoxen, PictureBoxen etc. autonome Fenster darstellen. Ausnahmen bilden hier hausbackene Controls wie Labels, Shapes, Images etc., sie gehören nicht dazu.

Das folgende Beispiel demonstriert die einfache Verwendung von Fensterhandles zweier Formulare, um deren Schnittpunkte, bei eventuell gegebener Überlagerung ihrer Umrisse, zu ermitteln:

Fensterhandles, Schnittpunktbetrachtung [Beispiel 12]

```
Option Explicit
```

```
Private Declare Function GetWindowRect Lib "user32.dll" (ByVal _  
    hwnd As Long, lpRect As RECT) As Long
```

```
Private Declare Function IntersectRect Lib "user32.dll" _  
    (lpDestRect As RECT, lpSrc1Rect As RECT, lpSrc2Rect As _  
    RECT) As Long
```

```
Private Type RECT  
    Left As Long  
    Top As Long  
    Right As Long  
    Bottom As Long  
End Type
```

```
Private Sub Form_Load()  
    Form2.Show  
    Timer1.Interval = 50  
    Timer1.Enabled = True  
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)
    Unload Form2
End Sub
```

```
Private Sub Timer1_Timer()
    Dim RectIntersect As RECT
    Dim RectForm1 As RECT
    Dim RectForm2 As RECT
    Dim Result As Long

    Call GetWindowRect(Form1.hwnd, RectForm1)
    Call GetWindowRect(Form2.hwnd, RectForm2)

    Result = IntersectRect(RectIntersect, _
        RectForm1, RectForm2)

    If Result <> 0 Then
        Form1.Caption = "Schnittpunkt!"
        Form2.Caption = "Schnittpunkt!"

        Label1.Caption = RectIntersect.Left
        Label2.Caption = RectIntersect.Top
        Label3.Caption = RectIntersect.Right
        Label4.Caption = RectIntersect.Bottom

    Else
        Form1.Caption = "Nebeneinander"
        Form2.Caption = "Nebeneinander"

        Label1.Caption = ""
        Label2.Caption = ""
        Label3.Caption = ""
        Label4.Caption = ""
    End If
End Sub
```

Gerätekontexte [DeviceContexts]

Ein Gerätekontext ist dem Erscheinen nach einem Handle sehr ähnlich. Genau wie sein großer Verwandter findet seine Definition und Organisation hinter den Kulissen statt, so daß wir auch hier wieder keinen direkten Einfluß nehmen können. Wie der Name vielleicht bereits erahnen läßt, steht der Gerätekontext in Zusammenhang mit einer physikalischen Hardware wie z.B. dem Monitor, der Tastatur oder dem Drucker. Ein solcher Kontext ist recht hilfreich, da er einer Anwendung gestattet die unterschiedlichsten Modelle und Marken eines Gerätetyps, beispielsweise Drucker, in ein und derselben Art und Weise zu handhaben. Praktisch bedeutet dies, daß eine Anwendung sich nicht weiter darum scheren muß, ob nun eine Laserdrucker der Marke HP oder ein Tintenstrahldrucker von Canon an den Rechner angeschlossen ist; der Gerätekontext macht sie (so gut wie) gleich.

Wie bereits Eingangs angedeutet, kann eine Anwendung den Gerätekontext nicht direkt beeinflussen, sondern ist hier wieder auf die Bereitstellung eines Handles angewiesen. Zu beachten ist hier, daß das Handle eines Gerätekontextes [hDC] funktional nicht dasselbe ist wie das eines Fensterhandles [hwnd], ansonsten handelt es sich aber bei beiden um einen vom System reservierten eindeutigen 32-Bit Wert.

Wohlüberlegt sind Fenster im Windowssinne auch Geräte, und verfügen daher über einen eigenen Kontext und somit über ein Handle. Dadurch ist es mit Hilfe der entsprechenden API-Funktionen möglich die grafische Oberfläche eines Fensters zu ändern und zu manipulieren.

Wie der Zufall es so will, greift uns Visual Basic auch hier wieder netterweise unter die Arme, indem es uns, genau wie bei der hwnd-Property, auch für den Gerätekontext das benötigte Handle unverbindlich zur Verfügung stellt. Das macht uns alles weitere recht leicht.

Das folgende Beispiel zeigt wie mit einfachen Mitteln, kleinere Hintergrundgrafiken Grafiken derart nebeneinander auf eine Formular gezeichnet werden können, daß der Eindruck eines geschlossenen Bildes entsteht. Hierbei wird mittels der API-Funktion **BitBlt** die jeweilige Kachel direkt in den Gerätekontext des Fensters kopiert:

Gerätekontext, Kacheln von Mustern [Beispiel 13]

Option Explicit

```
Private Declare Function BitBlt Lib "gdi32" (ByVal hDestDC As _
    Long, ByVal x As Long, ByVal y As Long, ByVal nWidth As _
    Long, ByVal nHeight As Long, ByVal hSrcDC As Long, _
    ByVal xSrc As Long, ByVal ySrc As Long, ByVal dwRop As _
    Long) As Long
```

```
Const SRCCOPY = &HCC0020
```

```
Dim BackGnr%
```

```
Private Sub Form_Load()
```

```
    Dim x&
```

```
    For x = 0 To Picture1.UBound
```

```
        Picture1(x).ScaleMode = vbPixels
```

```
        Picture1(x).Picture = LoadPicture(App.Path & "\Pic" & _
            CStr(x + 1) & ".jpg")
```

```
        Picture1(x).Refresh
```

```
    Next x
```

```
    Me.Picture = Picture1(0).Picture
```

```
    Form1.ScaleMode = vbPixels
```

```
    Me.Refresh
```

```
End Sub
```

```
Private Sub Form_Paint()
    Call TileForm(BackGNr)
End Sub
```

```
Private Sub Command1_Click(Index As Integer)
    BackGNr = Index
    Me.Picture = Picture1(BackGNr).Picture
End Sub
```

```
Private Sub TileForm(ByVal Pattern&)
    Dim x&, y&, Dx&, Dy&

    Dx = Picture1(Pattern).ScaleWidth
    Dy = Picture1(Pattern).ScaleHeight

    For x = 0 To Me.ScaleWidth Step Dx
        For y = 0 To Me.ScaleHeight Step Dy
            Call BitBlt(Me.hDC, x, y, Dx, Dy, _
                Picture1(Pattern).hDC, 0, 0, SRCCOPY)
        Next y
    Next x
End Sub
```

Flags

Flags [Flaggen] kommen als benannte numerische Konstanten daher und stehen für einen bestimmten Zustand eines bestimmten Kontextes. Solche Kontexte können z.B. Fensterstile, oder Schreib- und Leserechte in der Registry sein. Hier ein paar Flags für den Windowsstyle, der das grundsätzliche Aussehen von Fenstern bestimmt:

```
Const WS_TABSTOP = &H10000
Const WS_MINIMIZEBOX = &H20000
Const WS_THICKFRAME = &H40000
Const WS_SYSMENU = &H80000
```

Jede Konstante steht hier für einen ganz bestimmten Fensterstil. Das Setzen des Flags `WS_TABSTOP` entspricht der Visual Basic Property `Control.TabsStop = True`. `WS_MINIMIZEBOX` erlaubt das Rück- und Setzen des Minimieren Buttons, `WS_THICKFRAME` hat Einfluß auf die sichtbare Rahmenbreite eines Fensters und `WS_SYSMENU` kontrolliert das Vorhandensein eines Systemmenüs etc.

Wenn wir den Wert eines solchen Flags im Form eines Bitmusters näher betrachten, stellen wir fest, daß immer nur jeweils ein Bit gesetzt ist. Unter dieser Voraussetzung und dem Umstand, daß sich Flags zu 99,9% als 32-Bit Wert repräsentieren, können wir folgern, daß es maximal 32 verschiedene Flags pro Kontextgruppe geben kann.

Mit der Funktion [SetWindowLong](#) lassen sich die benannten Stile, bei bekannt sein des `hWnds` eines beliebigen Fensters, ändern. Was ist jetzt zu tun, wenn z.B. dieser API nicht nur ein Stile sondern eine Kombination der verschiedenen Darstellungsarten übergeben werden soll? Denkbar wäre z.B. ein Fenster mit einem dicken Rahmen, ohne Kontextmenü und Minimieren-Schaltfläche. Die Deklaration der [SetWindowLong](#) erlaubt aber nur die Angabe eines einzigen 32-Bit Werts für den Stil.

Da, wie bereits oben angeschnitten, jeder Stil ein unverwechselbares Bitmuster verfügt, müssen wir die Werte mit einander zu einem 32-Bit Wert verknüpfen. Dies machen wir mit dem logischen Operator `Or` [logische Oder Verknüpfung]:

```
Const WS_OURSTYLE = WS_MINIMIZEBOX Or WS_THICKFRAME Or WS_SYSMENU
```

Die Bitmuster werden somit übereinander gelegt und zu einem einzigen Wert vereint. Folgendes Beispiel nutzt dieses Vorgehen um Fensterstile zu ändern die unter VB normalerweise schreibgeschützt sind:

Fensterstile, Flags verknüpfen [Beispiel 14]

```
Option Explicit
```

```
Private Declare Function SetWindowLong Lib "user32" Alias _
    "SetWindowLongA" (ByVal hWnd As Long, ByVal nIndex _
    As Long, ByVal dwNewLong As Long) As Long
```

```
Private Declare Function GetWindowLong Lib "user32" Alias _
    "GetWindowLongA" (ByVal hWnd As Long, ByVal nIndex _
    As Long) As Long
```

```
Private Declare Function SetWindowPos Lib "user32" (ByVal _
    hWnd As Long, ByVal hWndInsertAfter As Long, ByVal _
    x As Long, ByVal y As Long, ByVal cx As Long, _
    ByVal cy As Long, ByVal wFlags As Long) As Long
```

```
Private Declare Function GetWindowRect Lib "user32" (ByVal _
    hWnd As Long, lpRect As Rect) As Long
```

```
Private Type Rect
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
```

```
Const SWP_FRAMECHANGED = &H20
Const GWL_STYLE = (-16)
Const WS_MAXIMIZEBOX = &H10000
Const WS_MINIMIZEBOX = &H20000
Const WS_THICKFRAME = &H40000
Const WS_SYSMENU = &H80000
Const WS_HSCROLL = &H100000
Const WS_VSCROLL = &H200000
Const WS_BORDER = &H800000
```

```
Private Sub Check1_Click(Index As Integer)
    Select Case Index
        Case 0: Call SetForm(WS_MAXIMIZEBOX, False)
                Call SetForm(WS_MINIMIZEBOX, True)
        Case 1: Call SetForm(WS_SYSMENU, True)
        Case 2: Call SetForm(WS_VSCROLL, True)
                Call SetForm(WS_HSCROLL, True)
        Case 3: Call SetForm(WS_BORDER, True)
        Case 4: Call SetForm(WS_THICKFRAME, True)
    End Select
End Sub
```

```
Private Sub SetForm(ToggleStyle&, FRefresh As Boolean)
    Dim lngStyle As Long, R As Rect

    lngStyle = GetWindowLong(Me.hWnd, GWL_STYLE)
    If (lngStyle And ToggleStyle&) Then
        lngStyle = lngStyle - ToggleStyle&
    Else
        lngStyle = lngStyle Or ToggleStyle&
    End If
    Call SetWindowLong(Me.hWnd, GWL_STYLE, lngStyle)
    Call GetWindowRect(Me.hWnd, R)
    Call SetWindowPos(Me.hWnd, 0, R.Left, R.Top, _
        R.Right - R.Left, R.Bottom - R.Top, _
        SWP_FRAMECHANGED)
End Sub
```

Die SendMessage

Sie ist ein universelles Unikum, anders kann ich sie nicht bezeichnen. Sie regelt unter Windows in Bezug auf Controls einen wesentlichen Teil des Nachrichtenverkehrs.

Hierzu müssen wir wissen, daß unter Windows eine recht anschauliche Menge fertiger Controls zur Verfügung stehen. Wir kennen sie bereits alle, denn unter VB sind sie ebenso zugegen. Angefangen mit dem einfachen CommandButton über TextBoxen, hinauf zum Fenster an sich, bis hin zum ListView und TreeView. Letztere gehören zwar nicht unbedingt zu den Standardkomponenten, sind aber bei Vorhandensein der comctl32.dll verfügbar.

Zurück zur [SendMessage](#), wie der Name bereits verrät, sendet sie Nachrichten. Die Frage ist, an wen und vor allem, welche Arten von Nachrichten. Ersteres ist leicht beantwortet, alles was über einen hWnd verfügt, also Fensterhandle, ist auch in der Lage durch die [SendMessage](#) vermittelte Nachrichten zu empfangen. Dazu gehören sämtliche Eingangs erwähnten Windowscontrols. Ausgegrenzt sind grundsätzlich Pseudocontrols wie z.B. Labels, Shapes und Timer.

Kommen wir zum Was. Nun ja halt Nachrichten ... was wir unter VB als Eigenschaften und Events kennen, nennt der C-Programmierer gemeinhin Messages, sprich eben Nachrichten. Das bedeutet also, daß was wir in VB in Bezug auf Controls mit unseren Eigenschaften und Methoden anstellen, sich meist auch alternativ direkt über eine äquivalente Nachricht bewirken läßt. Für Ereignisse trifft das noch viel stärker zu.

Nachrichten sind 32-Bit Werte. Sie werden in benannten Konstanten abgelegt und zerfallen aus Sicht des APIs in zwei Gruppen, zum einen die reinen Windowsnachrichten zum anderen in die controlspezifischen Nachrichten. Erstere besitzen den Präfix WM_ aller weiteren einen dem Namen des Controls entsprechenden Präfix, wie z.B. LB_ für ListBox, CB_ für ComboBox usw. Die meisten Controls vereinigen aber beide Nachrichten-Gruppen in sich.

Sie finden die gängigsten Nachrichten in Ihrem API-Viewer unter der Auswahl "Konstanten", hier ein paar Beispiele, bezogen auf die ListBox:

```
Const LB_GETTEXT = &H189
Const LB_GETTEXTLEN = &H18A
Const LB_GETCURSEL = &H188
Const LB_ADDSTRING = &H180
Const LB_INSERTSTRING = &H181
Const LB_DELETESTRING = &H182
```

Die Bezeichnung hinter dem Präfix `LB_` läßt Rückschlüsse auf die Wirkung der einzelnen Nachrichten auf das angesprochene Control zu. So dient `LB_GETTEXT` dem Auslesen des Textes eines ListBoxen-Eintrages, `LB_ADDSTRING` ist das Äquivalent zu `List1.AddItem` etc. Schauen wir uns jetzt einmal die Deklaration der `SendMessage` an:

```
Private Declare Function SendMessage Lib "user32" Alias _
    "SendMessageA" (ByVal hWnd As Long, ByVal wParam _
    As Long, ByVal lParam As Long, lParam As Any) _
    As Long
```

Auf den ersten Blick eine ganz normale API-Funktion. Der erste Parameter `hWnd` bezeichnete das Handle des anzusprechenden Controls [mittlerweile wissen wir ja, daß fast jedes Control ein solches Handle besitzt], der zweite `wMsg` nimmt die zu sendende Nachricht auf. Bei Betrachtung der beiden letzteren stellen wir jedoch fest, daß deren namentliche Benennung wenig Rückschlüsse auf ihre Bedeutung zuläßt. Sie werden jetzt verzweifelt auflachen, aber was für die beiden Parameter einzusetzen ist, läßt sich überhaupt nicht eindeutig erklären. Das ist natürlich ein schwerverdaulicher Hammer!

Die Lösung des Rätsels besteht darin, daß diese zwei illustren Knilche abhängig sind von dem jeweils verwendeten Nachrichtentyp. Kurzum: Andere Nachricht, andere Bedeutung. Berücksichtigt man, daß es unter Windows einige 1000 verschiedener Nachrichten gibt, könnte man mutlos werden. Aber keine Sorge auswendig lernen muß sie niemand, hierfür gibt es Nachschlagewerke und weitere kleine Helferlein. Sehen Sie hierfür in das weiter unten stehende Kapitel "*Information ist alles*".

Im folgenden finden Sie ein Beispiel, welches aufzeigt, wie einige der aus VB bekannten Eigenschaften und Methoden einer ListBox durch den bloßen Aufruf der `SendMessage` im gewohnten Maße geändert bzw. manipuliert werden können:

`SendMessage, ListBox per API handeln [Beispiel 15]`
`Option Explicit`

```
Private Declare Function SendMessage Lib "user32" Alias _
    "SendMessageA" (ByVal hWnd As Long, ByVal wParam _
    As Long, ByVal lParam As Long, lParam As Any) _
    As Long
```

```
Const LB_GETCOUNT = &H18B
Const LB_GETTEXT = &H189
Const LB_GETTEXTLEN = &H18A
Const LB_GETCURSEL = &H188
Const LB_ADDSTRING = &H180
Const LB_INSERTSTRING = &H181
Const LB_DELETESTRING = &H182
```

`'ListBox mit Anfangswerten füllen`

```
Private Sub Form_Load()
    List1.AddItem "Bohnen"
    List1.AddItem "Erbsen"
    List1.AddItem "Möhren"
    List1.ListIndex = 1
End Sub
```

`'Inhalt von List1 in List2 kopieren`

```
Private Sub Command1_Click()
    Dim lCnt As Long, x As Long

    'Anzahl der Einträge in List1 feststellen
```

```

lCnt = SendMessage(List1.hWnd, LB_GETCOUNT, 0, 0)

'Alle Einträge der Reihe nach auslesen und in
'List2 einfügen
If lCnt > 0 Then
    For x = 0 To lCnt - 1
        List2.AddItem GetListBoxEntry(List1.hWnd, x)
    Next x
End If
End Sub

```

```

'Diese Funktion liest einen nach dem Index benannten
'Eintrag aus
Private Function GetListBoxEntry(hWnd&, LbItem&) As String
    Dim L As Long, Buffer As String

    'Textlänge erfassen
    L = SendMessage(hWnd, LB_GETTEXTLEN, LbItem, 0)

    'Puffer anlegen
    Buffer = Space$(L + 1)

    'Text holen, Achtung -> ByVal
    L = SendMessage(hWnd, LB_GETTEXT, LbItem, ByVal Buffer)

    'Ereignis zurechtstutzen und zurückgeben
    GetListBoxEntry = Left$(Buffer, L)
End Function

```

```

'Auslesen des jeweils selektierten ListIndizes
Private Sub List1_Click()
    Dim LbIdx As Long

    'Index feststellen
    LbIdx = SendMessage(List1.hWnd, LB_GETCURSEL, 0, 0)
    If LbIdx <> -1 Then
        'Text des Indizes lesen
        Text1.Text = GetListBoxEntry(List1.hWnd, LbIdx)
    Else
        'Kein Index vorhanden
        Text1.Text = ""
    End If
End Sub

```

```

'Eintrag löschen
Private Sub Command2_Click()
    Dim LbIdx As Long

    'Index feststellen
    LbIdx = SendMessage(List1.hWnd, LB_GETCURSEL, 0, 0)
    If LbIdx <> -1 Then
        'Text des Indizes lesen
        Call SendMessage(List1.hWnd, LB_DELETESTRING, _
            LbIdx, 0&)
    End If
End Sub

```

```
        'Markierung auf den letzten Eintrag setzen
        List1.ListIndex = List1.ListCount - 1
    End If
End Sub
```

```
'Eintrag hinzufügen
Private Sub Command3_Click()
    'Text anhängen, Achtung ByVal
    Call SendMessage(List1.hWnd, LB_ADDSTRING, 0&, _
        ByVal ("Test, anhängen"))
    List1.ListIndex = List1.ListCount - 1
End Sub
```

```
'Eintrag einfügen
Private Sub Command4_Click()
    Dim LBidx As Long

    'Index feststellen
    LBidx = SendMessage(List1.hWnd, LB_GETCURSEL, 0, 0)
    If LBidx <> -1 Then
        'Text einfügen, Achtung ByVal
        Call SendMessage(List1.hWnd, LB_INSERTSTRING, LBidx, _
            ByVal ("Test, einfügen"))
        List1.ListIndex = LBidx
    End If
End Sub
```

```
'List2 löschen
Private Sub Command5_Click()
    List2.Clear
End Sub
```

Callbacks

Sie gestatten dem API eine gewisse Interaktivität mit der eigenen Anwendung zu entwickeln. Im Wesentlichen, wird in einer Anwendung nach vorgegebenen Bedingungen eine Hüllfunktion angelegt, die das API dann bei Eintreffen gewisser Umstände unmittelbar aufruft. Ihr Programm muß also die Funktion gemäß den Wünschen des APIs zur Verfügung stellen. Weiterhin muß dem API mitgeteilt werden, an welcher Speicheradresse die eingerichtete Callback aufrufbar ist. Das Ermitteln der Adresse gestaltet sich mit Hilfe des ab VB5 zur Verfügung stehenden `AddressOf` Operators relativ einfach:

```
FunctionAddress = AddressOf(CallBackFunction)
```

Da der `AddressOf` Operator unter VB nur in Standardmodulen syntaktisch korrekt plaziert werden kann, bedingt dies, daß Callbacks auch nur eben in solchen Modulen zum Einsatz kommen können.

Bevor der erste Aufruf der Callback stattfinden kann, ist dem API zuvor über einen separaten Initialisierungsaufruf die mit `AddressOf` ermittelte Adresse mitzuteilen. Hierfür steht in der Regel eine eigene API-Funktion zur Verfügung. Nebst der aufzurufenden Adresse, besteht oftmals die Möglichkeit einen frei definierbaren Wert mit zu übergeben. In einem solchen Falle handelt es sich meist um sogenannte Enummerationen, sprich Aufzählungen. Diese bilden auch den eigentlichen Schwerpunkt der vorhandenen Callback-Funktionen. Praktisch sieht das so aus, daß nach erfolgter Initialisierung, das API für jeden Punkt einer zu ermittelnden Sammlung die Callback einzeln aufruft. Daher auch der Name Enummeration. Gilt es beispielsweise eine Gruppe mit 10 Mitgliedern über eine Callback zu nummerieren, so wird die Rückroutine auch 10 mal vom API angesprochen. Es sei denn, die Callback, sie ist ja auch eine Funktion und verfügt daher über die Möglichkeit einen Wert zurückzuliefern, übergibt zwischenzeitig dem API nach ihrer jeweiligen Beendigung ein definiertes Abbruchzeichen, wie zum Beispiel den Wert 0. Dieser Umstand deutet dem API dann an, daß die Fortsetzung der Enummeration nicht mehr erwünscht ist, weitere Aufrufe unterbleiben daraufhin.

Wie Eingangs bereits angeschnitten, gibt es bei der Initialisierung des Callback-Procederes oftmals die Möglichkeit einen frei wählbaren `Long`-Wert mit eingehen zu lassen. Dieser spezifische Wert, wird dann bei jedem Aufruf der Callback an diese mitübergeben. Das kann z.B. komfortabel dazu genutzt werden um ein und dieselbe Callback von verschiedenen Punkten des Programms heraus zu aktivieren, so daß in der Callback selbst, durch Unterscheidung des zusätzlichen Parameters, kontextgemäß reagiert werden kann.

Der Konstruktor einer Callback an sich, ist ebenfalls recht unkritisch aufgebaut. In der Regel folgt er folgendem Schema:

```
Public Function EnumProc(ByVal Information As Long, _
                        ByVal lParam As Long) As Long
    ' ...
    ' ...

    EnumProc = 1
End Function
```

Die Variable `Information` ist der eigentlich gewünschte Wert, der beim Aufruf durch das API an die Callback übergeben wird, `lParam` der eben besprochene Zusatzparameter für die Kontext-Unterscheidung. `EnumProc = 1` bewirkt, daß das Fortfahren der Enummeration erwünscht ist, gäbe die Funktion hier einen anderen Wert zurück, würde die Aufzählung seitens des APIs abgebrochen. Angemerkt sei noch, daß eine Callback grundsätzlich als `Public` deklariert sein muß.

Zu beachten gilt noch, daß der vom API an den Prozedurkopf übergebenen Werte auch ein Zeiger sein kann, hier gelten dann die selben Bedingungen und Methodiken, die allgemein im Umgang mit Zeigern unter VB zu berücksichtigen sind. Im speziellen ist der Einsatz der `CopyMemory` wieder unabdingbar. Das folgende Beispiel zeigt hingegen den unproblematischen Umgang mit einer Enummeration, es werden alle `hWnds` [Fensterhandles] der Child-Windows, die auf einem Parent-Formular hocken, aufgezählt:

Callbacks, Enumeration [Beispiel 16]

```
Public Child() As Long
```

```
Public Function Init(hwnd As Long)
    Dim x As Long
    ReDim Child(0 To 0)

    Call EnumChildWindows(hwnd, AddressOf EnumChilds, 0)

    If UBound(Child) > 0 Then
        Debug.Print "-----"
        For x = 0 To UBound(Child) - 1
            Debug.Print Child(x)
        Next x
    End If
End Function
```

```
Public Function EnumChilds(ByVal ChWnd&, ByVal lParam&) As Long
    Child(UBound(Child)) = ChWnd
    ReDim Preserve Child(0 To UBound(Child) + 1)
    EnumChilds = 1
End Function
```

Die Enumeration ist also die denkbar unkritischste Version einer Callback-Funktion. Wesentlich riskanter wird es bei Verwendung von Techniken wie Subclassing und Hooking. Hier dient die Callback als direkte Filterschnittstelle im Nachrichtenstrom zwischen Windows und VB. Im Extremfall gilt es dort hunderten von Nachrichten pro Sekunde beizukommen. Insgesamt ein starkes und sehr manipulatives Werkzeug, dessen Erläuterung an dieser Stelle aber den Rahmen sprengen würde. Daher verweise ich Sie lieber auf einen weiteren Artikel namens [Subclassing leicht gemacht](#) [Artikel 11]

SafeArrays

Die oft einzig bekannte Technik unter VB Bitmaps pixelgenau zu manipulieren, sind die beiden sehr langsamen Methoden .Pset und .Point. Dabei steht uns auch in VB eine relativ einfache Möglichkeit, die das Einlesen von Bitmaps in ein Array innerhalb von Mikrosekunden bewerkstelligt, zur Verfügung. Das dürfte selbst in Assembler nur unwesentlich fixer gehen. Ist die Grafik erstmal im Feld gespeichert, lassen sich mit dieser Technik fast in Echtzeit, komplexere Operationen auf die Bitmap anwenden. Dieses Kapitel vermittelt das hierfür relevante Wissen, als auch die speichertechnischen Hintergründe.

Arrays unter Visual Basic besitzen genauso wie Strings einen Deskriptor. Dieser wird benötigt um die, im Vergleich zu anderen Programmiersprachen, vorherrschende hohe Dynamik von Feldern verwaltbar zu machen. So ist das dynamische Verschieben von Felder-ober- und Untergrenzen zur Laufzeit für den VB-Programmierer sehr praktisch und auch selbstverständlich, wie folgendes Beispiel zeigt:

```
Private Sub Command1_Click()
    Dim Feld() As Long

    ReDim Feld(0 To 20)
    ReDim Feld(-10 To 50)
End Sub
```

Dies wird nur möglich durch den besagten Deskriptor. Wir können uns ihn als einen Speicherbereich mit vordefiniertem Aufbau vorstellen. Am leichtesten läßt er sich als Kombination zweier Strukturen darstellen. Die folgende erste, ist Teil der zweiten und beschreibt die Arraygrenzen:

```
Private Type SAFEARRAYBOUND
    cElements As Long
    lLbound As Long
End Type
```

Beide Members der Struktur sind vom Typ `Long` und umfassen je 4 Byte. `cElements` stellt dabei die Gesamtzahl der im Feld enthaltenen Elemente dar. `lLbound` beschreibt die Untergrenze, also den Index des ersten Feldelementes. Für die Zeile `Dim Feld(-10 To 50)` stände demnach in `lLbound` `-10` und in `cElements` der Wert `61`.

Schauen wir uns jetzt die zweite Struktur an, in der `SAFEARRAYBOUND` ein Mitglied neben anderen ist:

```
Private Type SAFEARRAY1D
    cDims As Integer
    fFeatures As Integer
    cbElements As Long
    cLocks As Long
    pvData As Long
    Bounds(0 To 0) As SAFEARRAYBOUND
End Type
```

- `.cDims` umfaßt zwei Bytes und benennt die Dimensionen. Bei einem eindimensionalen Feld wie `Dim Feld(0 To 9)` enthält diese Variable den Wert `1`, für eine Deklaration wie `Dim Feld(0 To 9, 0 To 4)` den Wert `2` usw.
- `.fFeatures` ebenfalls 2 Byte lang, gibt Auskunft über den im Array verwendeten Datentyp. Für rein numerische Datentypen ist `fFeatures` immer `Null`, interessant wird es bei anders gearteten Variablen. Für die verschiedensten Arraytypen gibt es je ein Flag in Form einer Konstante.

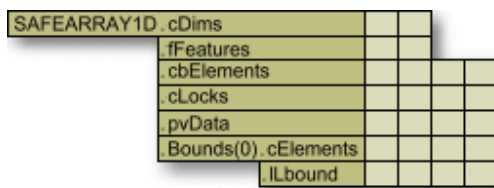
Konstanten Tabelle für `.fFeature`

Konstante	Wert	Beschreibung
FADF_AUTO	&H1	Das Array liegt im Stack
FADF_STATIC	&H2	Ein statisches Array
FADF_EMBEDDED	&H4	Das Feld ist in einer Struktur eingebettet
FADF_FIXEDSIZE	&H10	Die Grenzen des Arrays sind nicht änderbar
FADF_RECORD	&H20	Das Array enthält Records
FADF_HAVEIID	&H40	Array über eine IID-Schnittstelle identifizierbar ist
FADF_HAVEVARTYPE	&H80	Feld des Typs <code>VT</code>
FADF_BSTR	&H100	String-Array
FADF_UNKNOWN	&H200	Array der Schnittstelle <code>IUnknown</code>
FADF_DISPATCH	&H400	Array der Schnittstelle <code>IDispatch</code>
FADF_VARIANT	&H800	Array ist vom Typ <code>Variant</code>

Anzumerken ist, daß es bei der Verwendung von Stringarrays, auf Grund der Visual Basic internen Umwandlungen von `BSTR` in `ANSI` bzw. `Unicode`, nicht direkt möglich ist an den Arrayzeiger zu gelangen. Bei Verwendung der undokumentierten Funktion `VarPtrArray` ist lediglich ein Zeiger auf die jeweilige interne Kopie des temporären `ANSI`- bzw. `Unicode`-Strings zu erhalten. Zur Lösung des Problems muß eigens eine Type-Library erstellt werden. Dies soll hier aber nicht weiter erörtert werden.

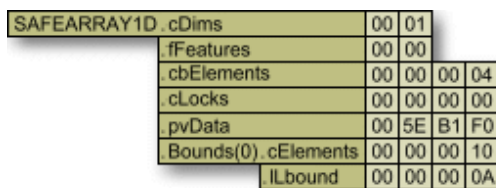
- **.cbElements** 4 Bytes, gibt Auskunft wieviel Speicherplatz ein einzelnes Element des Feldes benötigt. Für `Long`-Werte wären dies 4, für `Byte` 1, `Variant` 16 Byte etc.
- **.cLocks** 4 Bytes, Gibt den aktuellen Wert des Sperrzählers eines Arrays an. Dieser Sperrzähler tritt in Erscheinung, wenn z.B. ein in einer Struktur befindliches Array, welches mit einem `With MyStruct ... End With` Konstrukt umschlossen ist, geändert werden soll. In diesem Zustand ist das Feld gesperrt, Versuche sich darüber hinwegzusetzen führen zu einem Laufzeitfehler.
- **.pvData** Länge: 4 Bytes. Beinhaltet ebenfalls einen 4 Byte langer Zeiger der die eigentlichen Daten referenziert. Letztere sind losgelöst vom Deskriptor und befinden sich an beliebiger Stelle im Speicher. Dieser Umstand ist ein ganz besonderer Leckerbissen mit dem wir später noch ein paar nette Kunststückchen vollführen werden.
- **.Bounds(0 To 0)** Ist eine Struktur vom Typ `SAFEARRAYBOUND` mit zwei `Long`-Werten, also insgesamt 8-Bytes, die wie oben bereits angeschnitten, die Größe und die Untergrenze des Arrays aufnehmen. Der Parameter `(0 To 0)` deutet uns auch hier an, daß es sich um ein eindimensionales Feld handelt. Käme eine weitere Dimension hinzu, würde dort entsprechend `.Bounds(0 To 1)`, für drei Dimensionen `.Bounds(0 To 2)` usw. Vereinbarungsgemäß, hieße unsere Deskriptor-Struktur dann nicht mehr `SAFEARRAY1D` sondern `SAFEARRAY2D`, `SAFEARRAY3D`, `SAFEARRAY4D` ... `SAFEARRAYnD`.

Die Verteilung einer `SAFEARRAY1D` Struktur sähe als Speicherauszug bzw. –Belegung wie folgt aus:



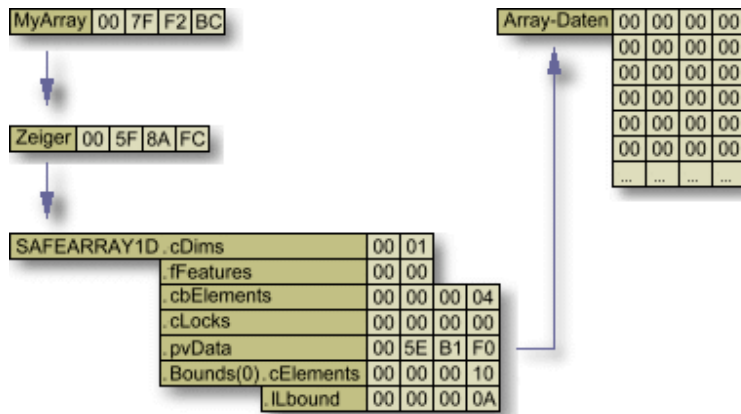
Die dunkelgrünen Felder zeigen die namentliche Benennung des zugehörigen Speicherplatzes, also werden hier die jeweiligen Member der Struktur dargestellt. Die leeren Kästchen dahinter bezeichnen je ein Byte. Daraus folgt, daß z.B. das Member `.fFeatures` 2 Byte und beispielsweise `.cLocks` hingegen 4-Bytes in Anspruch nimmt.

Schauen wir uns jetzt anhand einer einfachen Dimensionierung wie `Dim MyArray(10 To 25) As Long` den zugehörigen Deskriptor in hexadezimaler Darstellung im Speicher an:



`.CDims` beinhaltet wie zu erwarten den Wert 1, handelt es sich ja hier auch um ein eindimensionales Feld. `.fFeatures` steht auf Null, da es sich bei dem Feldtypen `Long` um einen rein numerischen handelt. Hätten wir anstatt dessen `Variant` gewählt, stände an dieser Stelle der Wert `&H800` [s.a. Tabelle oben]. `.cbElements` sagt uns, wie zu erwarten war, daß der verwendete Datentyp eines Elements [`Long`] pro Eintrag 4 Byte benötigt. `.cLocks` steht auf Null, da wir bis jetzt ja nur deklariert haben und keine der Sperrung förderlichen Operationen vorgenommen haben. `.pvData` ist als Zeiger auf die eigentlichen Felddaten ebenfalls vorhanden. `.Bounds(0).cElements` besitzt den hexadezimalen Wert `&H10`, entspricht 16, gleich bedeutend mit 16 Elementen [`10 To 25`]. Womit wir auch schon bei `.lLbound` wären, das die Untergrenze des Feldes korrekterweise auf 10 festlegt.

Jetzt beschleicht uns die Frage, wie ein so gearteter Deskriptor für uns zu nutzen ist. Als erstes ergibt sich hier das Problem, daß er über einen doppelten Zeiger referenziert wird. Dies stellt ersteinmal, auf Grund der normalerweise nicht vorhandenen Zeigerwerkzeuge unter VB, ein Problem dar. Zur Veranschaulichung hier der prinzipielle Aufbau des gesamten Arrangements als Speichermodell, mit seinen einzelnen Referenzen:



Um jetzt als ersten Schritt zu erfahren an welcher Stelle die Variable `MyArray` im Speicher abgelegt wurde, verwenden wir die in den Visual Basic Laufzeit-Bibliotheken vorhandene und durch Paul Wilde bekanntgemachte Funktion `VarPtrArray`. Ihre Deklaration sieht unter VB5 wie folgt aus:

```
Private Declare Function VarPtrArray Lib "msvbvm50.dll" _
    Alias "VarPtr" (Ptr() As Any) As Long
```

In VB6 ist sie entsprechend auf das folgende abzuwandeln:

```
Private Declare Function VarPtrArray Lib "msvbvm60.dll" _
    Alias "VarPtr" (Ptr() As Any) As Long
```

Angewandt auf `MyArray` erhalten wir den ersten Zeiger also mit :

```
PointerToPointer = VarPtrArray(MyArray)
```

Um dem zweiten, dem tatsächlich auf `SAFEARRAY1D` verweisenden, Zeiger habhaft zu werden, bemühen wir die **CopyMemory**, indem wir den Inhalt der durch den ersten Zeiger referenzierten 4 Bytes in eine weitere `Long`-Variable umkopieren:

```
Call CopyMemory(PointerToSafeArray, ByVal PointerToPointer, 4&)
```

Voilà, jetzt ist die Position des Deskriptors im Speicher bekannt. Durch das Anlegen einer gleichgearteten Struktur im Programm, kann nun von dort der Deskriptor einkopiert werden:

```
Call CopyMemory(SafeArray, ByVal PointerToSafeArray, Len(SafeArray))
```

Jetzt stehen alle Türen offen um beliebige Manipulationen an einem Array vornehmen zu können. Das folgende Beispiel, zeigt das gesamte Vorgehen nochmals ausführlicher. Es liest im wesentlichen den Deskriptor eines angelegten Arrays ein und gibt ihn benannt und formatiert in einer Textbox aus:

```
SafeArray ermitteln [Beispiel 17]
Option Explicit
```

```
Private Declare Function VarPtrArray Lib "msvbvm50.dll" _
    Alias "VarPtr" (Ptr() As Any) As Long
```

```

'Private Declare Function VarPtrArray Lib "msvbvm50.dll" _
'    Alias "VarPtr" (Ptr() As Any) As Long

Private Declare Sub CopyMemory Lib "kernel32" Alias _
    "RtlMoveMemory" (pDst As Any, pSrc As Any, _
    ByVal ByteLen As Long)

Private Type SAFEARRAYBOUND
    cElements As Long
    lLbound As Long
End Type

Private Type SAFEARRAY1D
    cDims As Integer
    fFeatures As Integer
    cbElements As Long
    cLocks As Long
    pvData As Long
    Bounds(0 To 0) As SAFEARRAYBOUND
End Type



---



Private Sub Command1_Click()
    Call GetSafeArray
End Sub



---



Private Sub GetSafeArray()
    Dim MyArray() As Long
    Dim SafeArray As SAFEARRAY1D
    Dim PointerToPointer As Long
    Dim PointerToSafeArray As Long

    ReDim MyArray(10 To 25)

    PointerToPointer = VarPtrArray(MyArray)
    Call CopyMemory(PointerToSafeArray, _
        ByVal PointerToPointer, 4&)

    Call CopyMemory(SafeArray, ByVal PointerToSafeArray, Len(SafeArray))

    Call DisplaySafeArray(SafeArray)
End Sub



---



Private Sub DisplaySafeArray(SA As SAFEARRAY1D)
    Dim H As String

    H = ".cDims = " & SA.cDims & vbCrLf
    H = H & ".fFeatures = " & SA.fFeatures & vbCrLf
    H = H & ".cbElements = " & SA.cbElements & vbCrLf
    H = H & ".cLocks = " & SA.cLocks & vbCrLf
    H = H & ".pvData = " & SA.pvData & vbCrLf
    H = H & ".Bounds(0).cElements = " & SA.Bounds(0).cElements & vbCrLf
    H = H & ".Bounds(0).lLbound = " & SA.Bounds(0).lLbound & vbCrLf
    Text1.Text = H
End Sub

```

Wie eben bereits schon angeschnitten, besteht die Möglichkeit Manipulationen am Deskriptor selbst vornehmen zu können. Wir können aber bereits schon auf einer viel früheren Ebene ansetzen. Normalerweise wird der Deskriptor durch einen Doppelzeiger referenziert. Denkbar wäre jetzt folgendes Szenario: Zwei Felder gleicher Größe, Grenzen und Datentyps aber verschiedenen Inhalts existieren friedlich nebeneinander her. Damit beide ihren Inhalt tauschen können, wäre für die reine VB-Lösung das Anlegen eines dritten, temporären Feldes gleicher Art nötig. Anschließend müßte in einer Schleife jedes Element einzeln umkopiert werden. Dieses Vorgehen ist recht mühselig und vor allem bei größeren Feldern sehr langsam. Eine Geschwindigkeitssteigerung ließe sich noch durch die Verwendung der **CopyMemory** erzielen, aber auch hier muß immer das gesamte Feld dreimal umgeschichtet werden.

Viel leichter und in erster Linie unschlagbar schneller ginge dies über das Tauschen der beiden auf die Deskriptoren zielenden Zeiger. Im Ausgangszustand verweist Zeiger1 auf das SafeArray1 und Zeiger2 auf SafeArray2. Durch eine kleine Manipulation, könnte man aber auch ohne weiteres Zeiger1 auf SafeArray 2 und Zeiger2 auf SafeArray1 verweisen lassen.

Hierzu ist lediglich der erste Zeiger des einen Deskriptor-Doppelpointers in einen temporären zwischenspeicher, der zweite in den ersten zu kopieren um abschließend den temporären in den zweiten zu übertragen. Unter der Berücksichtigung, daß der Zeiger auf einen Deskriptor lediglich 4 Byte umfaßt, werden also genau 12 Byte bewegt. Daß bedeutet das Austauschen zweier Arrays, egal ob 10 Bytes oder 30 MB groß, bedarf lediglich der Zeit, die für das Kopieren von 3 x 4 Bytes benötigt wird. Das folgende Beispiel verwendet diese Methode des Zeigertauschens:

SafeArray-Pointer tauschen [Beispiel 18]

Option Explicit

```
Private Declare Function VarPtrArray Lib "msvbvm50.dll" _
    Alias "VarPtr" (Ptr() As Any) As Long
```

```
'Private Declare Function VarPtrArray Lib "msvbvm60.dll" _
'    Alias "VarPtr" (Ptr() As Any) As Long
```

```
Private Declare Sub CopyMemory Lib "kernel32" Alias _
    "RtlMoveMemory" (pDst As Any, pSrc As Any, _
    ByVal ByteLen As Long)
```

```
Dim MyArray1() As Long
```

```
Dim MyArray2() As Long
```

```
Dim at() As Long
```

```
Private Sub Form_Load()
```

```
    Dim x As Long
```

```
        ReDim MyArray1(0 To 24)
```

```
        ReDim MyArray2(0 To 24)
```

```
        For x = 0 To 24
```

```
            MyArray1(x) = x
```

```
            MyArray2(x) = x * 100
```

```
        Next x
```

```
        Call DisplayArrays
```

```
End Sub
```

```
Private Sub Command1_Click()
```

```
    Call SwapDescriptors
```

```
End Sub
```

```
Private Sub SwapDescriptors()  
    Dim a1 As Long  
    Dim a2 As Long  
    Dim a3 As Long  
  
    a1 = VarPtrArray(MyArray1)  
    a2 = VarPtrArray(MyArray2)  
    a3 = VarPtrArray(at)  
  
    Call CopyMemory(ByVal a3, ByVal a1, 4)  
    Call CopyMemory(ByVal a1, ByVal a2, 4)  
    Call CopyMemory(ByVal a2, ByVal a3, 4)  
  
    Call DisplayArrays  
End Sub
```

```
Private Sub DisplayArrays()  
    Dim x As Long  
  
    List1.Clear  
    List2.Clear  
    For x = 0 To UBound(MyArray1)  
        List1.AddItem MyArray1(x)  
        List2.AddItem MyArray2(x)  
    Next x  
End Sub
```

Schnelle Bitmap-Operationen

Behalten wir die Sache mit den Deskriptoren des letzten Kapitels in Erinnerung und kommen jetzt kurz auf eine andere Thematik zu sprechen. Später werden wir dann beide Bereiche mühelos zusammenführen können.

Wird in einen Speicherbereich eine Bitmap als Datei eingeladen, so ist sie damit im Windowssinne ein Objekt und verfügt über ein Handle, das sogenannte Bitmaphandle. Dieses kann an die API-Funktion [GetObject](#) übergeben werden, um dadurch weitere Informationen über das Objekts zu erhalten. Da sich in VB hinter dem Wort `GetObject` bereits ein Befehl zur Ermittlung von ActiveX-Verweisen verbirgt, hat sich für die API-Funktion der Name [GetObjectAPI](#) eingebürgert.

In VB ist es möglich der Eigenschaft `.Picture` über den Befehl `LoadPicture` ein Bitmap-Objekt zuzuführen und damit beispielsweise dem Objekt `PictureBox` ein gültiges Objekthandle zuzuweisen. Somit kann dieses auch in der besagten Funktion zur Anwendung kommen.

Die Deklaration der Funktion lautet wie folgt:

```
Private Declare Function GetObjectAPI Lib "gdi32" Alias _
    "GetObjectA" (ByVal hObject As Long, ByVal nCount As _
    Long, lpObject As Any) As Long
```

- `hObject` Das Objekthandle, in unserem Falle die in eine `PictureBox` eingeladene Bitmap, referenziert durch den dem Objekt innewohnenden `Long`-Wert selbst.
- `lpObject` Zeiger auf einen Puffer der nach Aufruf der Funktion Informationen über das durch `hObject` bezeichnete Objekt zurückgibt.
- `nCount` Die Größe des mit `lpObject` festgelegten Puffers in Byte.

Der [Win32.hlp](#) ist zu lesen, daß für unsere Bitmap die Information in Form einer Struktur namens `BITMAP` zurückgegeben wird. Diese hat folgenden Aufbau:

```
Private Type BITMAP
    bmType As Long
    bmWidth As Long
    bmHeight As Long
    bmWidthBytes As Long
    bmPlanes As Integer
    bmBitsPixel As Integer
    bmBits As Long
End Type
```

Als Beschreibung der einzelnen Mitglieder ist zu entnehmen:

- `bmType` Spezifiziert den Bitmaptypen, dieses Member muß Null sein.
- `bmWidth` Die Breite der Bitmap in Pixel. Die Weite muß immer größer Null sein.
- `bmHeight` Die Höhe der Bitmap in Pixel. Die Höhe muß immer größer Null sein.
- `bmWidthBytes` Spezifiziert die Anzahl der Bytes pro ScanLine. Dieser Wert muß durch zwei teilbar sein, damit Windows Werte aus einem Array mit `word`-Ausrichtung annehmen kann.
- `bmPlanes` Anzahl der Farbebenen.
- `bmBitsPixel` Anzahl der Bits die benötigt werden um ein Pixel farblich anzeigen zu können.
- `bmBits` Ein Zeiger auf die Position der Bits einer Bitmap. `bmBits` muß ein 4-Bytelanger Pointer auf ein Feld mit Bytewerten sein.

Oha, sehr verdächtig, vor allem des letzte Member `bmBits`, läßt uns plötzlich wieder ein `SafeArray` denken. Zuerst einmal müssen wir aber die `GetObjectAPI` erfolgreich zur Anwendung bringen:

```
Private Sub GetBitmapStruct()  
    Dim Bmp As BITMAP  
  
    Picture1.Picture = LoadPicture(App.Path & "\Bild.jpg")  
    Call GetObject(Picture1.Picture, Len(Bmp), Bmp)  
End Sub
```

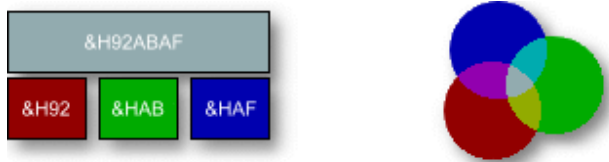
Betrachten wir nochmals das letzte Member der Bitmap-Struktur, dort steht:

Ein Zeiger auf die Position der Bits einer Bitmap. `bmBits` muß ein 4-Bytelanger Pointer auf ein Feld mit Bytewerten sein.

Was soviel bedeutet, daß hier ein Pointer abgelegt wird der auf den Datenblock eines Arrays vom Typ `Byte` zeigt. Wir haben also eine Struktur, die ein Member enthält, welches auf einen reinen Datenabschnitt im Speicher verweist. Die selbe Situation kennen wir bereits von den `SafeArrays`. Auch dort zeigt `pvData` lediglich auf einen Speicherblock. Wäre es jetzt nicht denkbar in Visual Basic ein `Bytearray` zu initialisieren und im Deskriptor das Mitglied `.pvData` so zu ändern, daß es genau auf den Speicherbereich der Bitmap verweist? Natürlich ist das denkbar, und zudem sogar auch machbar.

Damit wir genau wissen wie das `Bytearray` anzulegen ist, muß vorab exakt bekannt sein, wie die einzelnen Pixel einer Bitmap im Speicher arrangiert sind. Wir beschränken uns vorerst auf 24-Bit Bitmaps. Diese haben den entscheidenden Vorteil, daß für die Darstellung eines ihrer farbigen Pixel 3 x 8 Byte, also 24 Bit benötigt werden. Jedes Byte eines solchen Pixels bildet einen Farbanteil: Der zugehörige 24-Bit-Farbwert ist daher eine RGB-Zahl, die mit je einem Byte in Rot, Grün und Blau zerfällt.

Ein Beispiel:



Der obere Blau-Grauton [linke Abbildung] besitzt den 24-Bitwert `&H92ABAF`. Anhand der hexadezimalen Darstellung lassen sich gut die einzelnen Anteile erkennen, nämlich `&H92` für Rot, `&HAB` für Grün, sowie `&HAF` für Blau. Die rechte Abbildung beweist, daß sich durch Mischen der ermittelten Anteile wieder der originale Blau-Grauton ergibt.

In unserer Bitmap liegen also jeweils immer 3 Byte nebeneinander, die nach dem oben genannten Schema je ein Farbpixel bilden.

Würden wir in einem Grafikprogramm eine 24 Bit-Bitmap der Farbe weiß mit 4 x 4 Pixel kreieren, ließe sich ihr visueller Aufbau wie folgt vorstellen. Jedes Kästchen steht dabei für ein Byte, so daß sich pro Zeile ein Bedarf von 12 Byte, also insgesamt 48, ergibt:

	Pixel 1			Pixel 2			Pixel 3			Pixel 4		
Zelle 1	R0	G0	B0	R1	G1	B1	R2	G2	B2	R3	G3	B3
Zelle 2	R4	G4	B4	R5	G5	B5	R6	G6	B6	R7	G7	B7
Zelle 3	R8	G8	B8	R9	G9	B9	R10	G10	B10	R11	G11	B11
Zelle 4	R12	G12	B12	R13	G13	B13	R14	G14	B14	R15	G15	B15

Umgewandelt in ein zweidimensionales Bytearray wäre das erste sichtbare Pixel in der oberen linken Ecke mit $B(0,0)$, $B(0,1)$ und $B(0,2)$ zu adressieren. Das zweite Pixel hätte dann die Indizes $B(0,3)$, $B(0,4)$ und $B(0,5)$ usw.

Im Speicher sieht Organisation ein kleines bißchen anders aus. So liegen die Farbanteile nicht in der Reihenfolge R -> G -> B, sondern genau umgekehrt, nämlich in der Kette B -> G -> R vor. Weiterhin ist die gesamte Bitmap um die x-Achse gespiegelt, so daß das erste Byte der ersten Zeile das erste Byte der letzten Zeile einnimmt:

Zelle 4	B12	G12	R12	B13	G13	R13	B14	G14	R14	B15	G15	R15
Zelle 3	B8	G8	R8	B9	G9	R9	B10	G10	R10	B11	G11	R11
Zelle 2	B4	G4	R4	B5	G5	R5	B6	G6	R6	B7	G7	R7
Zelle 1	B0	G0	R0	B1	G1	R1	B2	G2	R2	B3	G3	R3
	Pixel 1	Pixel 2	Pixel 3	Pixel 4								

Möchten wir das erste sichtbare Pixel, wie oben, indizieren, geht der Ansatz diesmal über die Position der drei Byte in der unteren linken Ecke, also $B(3,0)$, $B(3,1)$ und $B(3,2)$. Damit es später nicht zu Überraschungen kommt, sollte dieser Aufbau stets präsent sein.

Zurück zur ursprünglichen Idee. Wir wollten ein Bytearray vorbereiten, dessen Deskriptor auf den in der BITMAP-Struktur referenzierten Speicherbereich weist. Dafür muß in erster Linie das Feld richtig dimensioniert werden. Da nicht nur die Arraygrenze sondern auch der Zeiger `.pvData` geändert werden muß, kann die BASIC Funktion `ReDim` nicht zum Einsatz kommen. Der Arraydeskriptor ist selbst zu konstruieren. Weil ein zweidimensionales Feld notwendig ist, sollte jetzt die Struktur `SAFEARRAY2D` verwendet werden:

```
Private Type SAFEARRAY2D
    cDims As Integer
    fFeatures As Integer
    cbElements As Long
    cLocks As Long
    pvData As Long
    Bounds(0 To 1) As SAFEARRAYBOUND
End Type
```

Durch

```
Call GetObject(Picture1.Picture, Len(Bmp), Bmp)
```

ist die `BITMAP`-Struktur zu erhalten. Nun wird der Deskriptor entsprechend zusammengesetzt:

```
With SafeArray
    .cDims = 2
    .cbElements = 1
    .fFeatures = 0
    .cLocks = 0
    .pvData = Bmp.bmBits

    .Bounds(0).lLbound = 0
    .Bounds(0).cElements = Bmp.bmHeight

    .Bounds(1).lLbound = 0
    .Bounds(1).cElements = Bmp.bmWidthBytes
End With
```

`.cDims` erhält den Wert 2, da es sich ja um ein zweidimensionales Feld handeln soll. Der Datentyp der Feldelemente ist `Byte`. Dieser benötigt pro Element nur je eine Speicherzelle, daher ist `.cbElements` auf 1 zu setzen. Der Datentyp `Byte` ist ein numerischer, woraus folgt, daß `.fFeatures` logischerweise den Wert 0 erhält. Sperrungen gibt es bisher keine, weshalb `.cLocks` auch gleich 0 ist.

Jetzt kommt der eigentliche Trick:

```
.pvData = Bmp.bmBits
```

`.pvData` würde normalerweise auf den von Visual Basic zugewiesenen Speicherblock des Bytearrays verweisen. `Bmp.bmBits` referenziert hingegen ebenfalls ein Bytearray, nämlich das der Bitmap. Indem `.pvData` jetzt `Bmp.bmBits` zugewiesen wird, ist erreicht, daß unser Array gleich der in der `PictureBox` befindlichen Bitmap ist. Also kurzum: Wenn der neue Deskriptor zugewiesen wird, befindet sich die gesamte Bitmap auf einen Schlag in einem Visual Basic Bytearray!

Doch weiter. Was jetzt noch fehlt sind die Arraygrenzen. Da ja ein zweidimensionales Feld konstruiert werden soll, sind auch zwei `Bounds` zu berücksichtigen. Damit die Sache zu Beginn so einfach wie möglich gehalten wird, erstellen wir ein nullbasierteres Array. Deshalb sind `.Bounds(1).lLbound` und `.Bounds(1).lLbound` auf Null zu setzen.

Das Array muß die Bedingung `BMP-Höhe_in_Pixel * (BMP-Breite_in_Pixel * 3)` erfüllen. Die Breite ist mit 3 zu multiplizieren, da eine 24-Bit Bitmap vorliegt und dort jedes Pixel 3 Byte benötigt. Die Höhe ist leicht ermittelt, liegt sie doch im Member `Bmp.bmHeight` der `BITMAP`-Struktur vor. `.BmWidth` ist zwar ebenfalls vorhanden, doch können die Operationen der Umrechnungen erspart bleiben, wenn wir uns die Beschreibung des Members `Bmp.bmWidthBytes` nochmals anschauen:

***bmWidthBytes** Spezifiziert die Anzahl der Bytes pro ScanLine. Dieser Wert muß durch zwei teilbar sein, damit Windows Werte aus einem Array mit `word`-Ausrichtung annehmen kann.*

Die Scanline ist in unserem Falle eine Zeile der Bitmap, also genau das was benötigt wird, daher können wir dieses Member zuweisen. Es ergibt sich für die `Bounds`:

```
.Bounds(0).lLbound = 0
.Bounds(0).cElements = Bmp.bmHeight

.Bounds(1).lLbound = 0
.Bounds(1).cElements = Bmp.bmWidthBytes
```

Jetzt gilt es nur noch den fertigen, neuen Deskriptor wie folgt anzubringen:

```
Call CopyMemory(ByVal VarPtrArray(MyArray), VarPtr(SafeArray), 4&)
```

Der Zeiger auf unsere gerade erst definierte Struktur, wird auf den ersten Zeiger des Doppelpointer kopiert. Das war alles. Durch das Umkopieren eines 4 Byte langen Pointers kann in Mikrosekunden eine Bitmap beliebiger Größe in ein Bytearray kopiert werden. Schneller geht es nicht. Aber es kommt noch besser.

Wir haben jetzt die Bitmap in ein Bytearray eingebettet und daher wirkt sich jede Änderungen eines Wertes im Feld direkt auf die Bitmap aus. Das heißt, ändern wir drei nebeneinander liegende Bytes mit je dem Wert `&HFF` auf den Wert `&H00`, wechselt auch das angesprochene Pixel seine Farbe unmittelbar von ehemals weiß in schwarz. Da wir mit der beschriebenen Technik an der `.Image`- und nicht der `.Picture`-Property der `PictureBox` werkeln, ist die Visualisierung der angewendeten Manipulationen mit `Picture1.Refresh` in den Vordergrund zu bringen.

Einmal als Array vorliegend, lassen sich mit dieser Methode in erstaunlichen Geschwindigkeiten Bilder fast [abhängig von der Bildgröße] in Echtzeit beliebig manipulieren. Es sei noch der Hinweis gegeben, daß nach Beendigung der grafischen Operationen mit der Zeile

```
Call CopyMemory(ByVal VarPtrArray(MyArray), 0&, 4&)
```

der Deskriptor des generierten Bytearray auf Null gesetzt werden sollte. Unter Win9x ist dies zwar überflüssig, bei NT-Systemen hingegen kann die nicht Verwendung dieser Zeile zu unangenehmen Nebenwirkungen führen.

Um die verschiedensten Effekte, wie Emboss, Ripple, Schärfe, Rotieren, sowie Ändern von Kontrast, Helligkeit und RGB-Verhältnissen, zu erzielen, schauen Sie bitte in die diesem Artikel beiliegende Beispiel-Sammlung [Bsp.19 bis Bsp. 24]. Die verwendete Methodik ist bei allen die gleiche, nämlich die hier besprochene. Es variieren lediglich die grafischen Algorithmen zwecks Erzielung der verschiedenen Effekte.

Nachteil dieser durch Matthew Curland und Francesco Balena erstmalig 1998 vorgestellten Zeigermethode, ist, daß nur Grafiken manipuliert werden können, die auch ein Bitmaphandle besitzen und das ist leider nicht immer gegeben. Ausweichmöglichkeit bietet das Arrangieren einer Grafik in einer DIB [Device Independent Bitmap]. Als Ergebnis liegt die Bitmap zwar wieder als Bytearray vor, hat aber nicht den harten und daher schnellen Kontakt zur Grafik. Für statische Bilder mag das keine Rolle spielen, bei raschem dynamischen Wechsel, sprich einer Animation, kann es aber schnell zu einer unflüssigen Darstellung kommen.

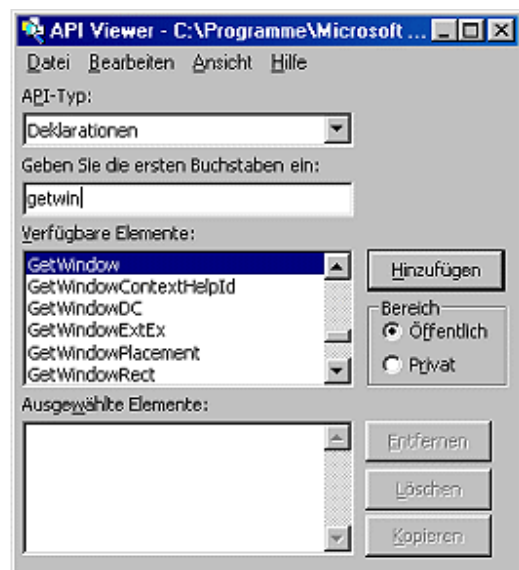
Information ist alles

Wie am Umfang dieses Artikels leicht erkennbar ist, umspannt das Thema API eine Unmenge von Möglichkeiten, Ausnahmen, Bedingungen etc. Unter Berücksichtigung der Tatsache, das es tausende von Funktionen, Typen und annähernd 100.000 Konstanten gibt, die zu allem Überfluß auch noch täglich erweitert werden, läßt sich vermuten, daß wir uns in einem riesigen Chaos befinden, welches ohne zusätzliche Hilfen wohl lange nicht mehr überschaubar ist. Oftmals treten sogar Widersprüche in den Erläuterungen des Herstellers selbst, nämlich Microsoft, auf. Angesichts dieser Fülle sind einige kleine bis große Helferlein unabdingbar.

Das in dieser Hinsicht wohl bekannteste Programm wird standardmäßig mit VB angeliefert. Der API-Viewer. Sollte er sich nicht in Nähe des VB-Eintrags Ihrer Startleiste herumtreiben, ist er im Verzeichnis "Microsoft Visual Studio\Common\Tools\Winapi\APILOAD.EXE" zu finden.

Er verfügt über einen schon ganz ansehnlichen Katalog gängiger API-Funktionen des täglichen Gebrauchs. Neben den Funktionen sind auch die üblichsten Konstanten und Strukturen enthalten. Besonders auszeichnend ist, daß sämtlicher Inhalt bereits der VB-Syntax angepaßt ist, so daß ein mühseliges Konvertieren von C-Deklarationen entfällt.

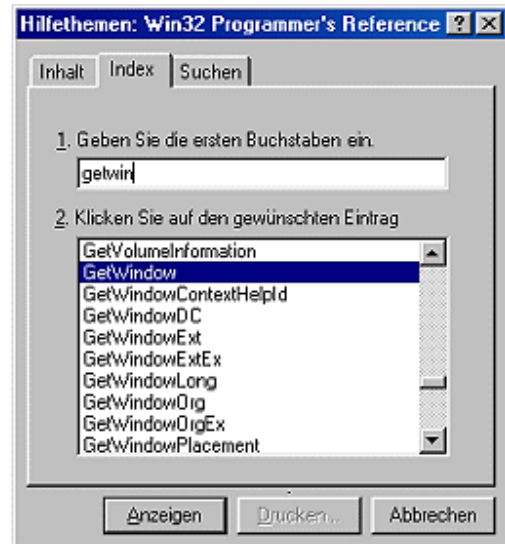
In seiner Auswahl kann beliebig gestöbert und mit einem Klick auch gesammelt werden, so daß nach Abschluß der Arbeiten, die gefundenen Informationen über die Zwischenablage direkt in VB eingefügt werden können.



Schwieriger wird es, wenn man im API-Viewer nicht fündig wird. Falls vorhanden sollte dann zu aller erst ein Blick in die MSDN [auch Online] geworfen werden, da hier manchmal fertige Visual Basic Beispiele angeboten werden, die im API-Viewer nicht vertretene Funktionen nutzen.

Sollte auch diese Suche erfolglos bleiben und das Internet ebenfalls keine weiteren Geheimnisse verraten, ist wohl oder übel auf C-basierte Quellen zurückzugreifen. Eine davon ist die bereits erwähnte MSDN. Ich selbst bevorzuge hierfür aber vor allem die [Win32.hlp](#). Ein als Helpfile aufgebautes Nachschlagewerk, mit knapp 20 MB Umfang und im Verhältnis zur MSDN doch recht handliches Ding. Es ist zwar letztendlich auch Bestandteil der MSDN aber durch seine Kompaktheit und themenspezifische Begrenzung angenehmer zu handhaben als seine große Schwester.

Wie schon erwähnt werden sämtliche Funktions-Deklarationen in C abgehalten. Gewisse Grundkenntnisse dieser Sprache sollten also schon vorhanden sein. Vorteilhaft ist vor allem, daß für jede Funktion, Struktur und auch Konstante eine knappe bis ausführliche Beschreibung, wenn auch in Englisch, vorliegt. Weiterhin stolpert man erfreulicherweise hin und wieder über kleine Grundlagentutorials. Das anfängliche Arbeiten mit der Hilfe ist vielleicht etwas verwirrend und gewöhnungsbedürftig, aber nach einer relativ kurzen Einarbeitungszeit, macht das Stöbern in ihr fast schon Spaß.



Alles Gute hat auch meist einen Haken. Der Schwerpunkt der [Win32.hlp](#) liegt eindeutig bei den Konstanten. Man kann es ihr nicht verdenken, wenn man weiß, daß C-Programmierer die Werte für Ihre Konstanten aus den zu jeder DLL vorliegenden Headerfiles beziehen. Wir VBler haben so etwas leider nicht. Kurzum, in der [Win32.hlp](#) erfährt man zwar fast alles über die Besonderheit und Funktion bestimmter Konstanten, nicht aber ihren Wert und der ist ja nun unabdingbar.

Aber was hindert uns daran einen verschämten Blick in die besagten Header unsere C-Kollegen zu werfen? Nichts, deshalb sollten wir dies bei Bedarf auch unbedingt tun. Sie befinden sich im Lieferumfang des Visual Studios, gesammelt in einem extra für diese Zwecke angelegten Ordner und sortiert in weiteren Unterordnern. Die Suche nach einer unbekanntem Konstante gestaltet sich zwar etwas martialisch, ist aber dennoch meist recht effektiv.

Suchen Sie hierzu im Windows-Explorer den besagten Header-Ordner auf, markieren Sie ihn und drücken sie die Tasten [Strg + f]. Ja Sie haben ganz recht gelesen, wir arbeiten mit dem windowseigenen Dateien-Suchen-Dialog. Da er auch über die Option der Volltextsuche verfügt, liegt nichts näher als diese auch für unsere Zwecke zu nutzen. War die Suche erfolgreich, wird in der Regel eine manchmal auch zwei Dateien gefunden. Diese, meist die größere, ist jetzt im WordPad zu öffnen. Eine erneute Suche im Text wird uns nun direkt zu unserer begehrten Konstante einschließlich ihres Wertes führen.

Subclassing, der geheime Nachrichtendienst

Subclassing ist eine Technik um ohne zusätzliche Komponenten an standardmäßig nicht zur Verfügung gestellte Ereignisse, sprich Nachrichten, zu gelangen. Grundsätzlich ist sie schnell zu erlernen und meines Erachtens derzeit für VB noch sehr wichtig. Dieser Artikel soll den Einstieg erleichtern, und die elementaren Dinge erläutern, die es hierbei zu berücksichtigen gilt.

Hinter den Kulissen eines Fensters, vom VB-Anwender gut abgeschirmt, wird so einiges geflüstert und getuschelt. Gemeint ist der Nachrichtenstrom unter Windows. Womit sich der C++ Programmierer täglich auseinandersetzt, bleibt uns unbedarften VBler auf längere Zeit indirekt verborgen. Indirekt deshalb, da wir ihn nur in Form der einschlägig bekannten Ereignisse nutzen. Ein einfaches Beispiel wäre das Click-Ereignis in einer `PictureBox`. Erfahrungsgemäß ausgelöst durch einen Klick, setzt es sich in Wirklichkeit aus zwei Nachrichten zusammen, nämlich dem Herunterdrücken und als Folge natürlich auch dem Lösen der Maustaste, also in der Summe wieder ein Klick.

Wie läuft das intern nun genau ab? Dazu müssen wir wissen, daß jedes Fenster eine Nachrichtenprozedur sein Eigen nennt. Der Begriff Fenster ist vielleicht etwas irreführend, da auch `CommandButtons` und `TextBoxen` etc. im Windowssinne zu dieser Gruppe gehören und ein sogenanntes `ChildWindow` darstellen. Allesamt besitzen je ein Fensterhandle, kurz `hWnd`. Ein solches Handle identifiziert ein Fenster im System eindeutig und wird niemals doppelt zugeteilt. Es sei denn, sein letzter Inhaber wurde geschlossen oder nach einem Reboot. [Siehe auch Kapitel "Handles"]

Die Handles werden dynamisch, nach Bedarf vergeben und es ist auszuschließen, daß Fenstern z.B. nach einem Neustart je ein und die selben Handles erhalten. Zum besseren Verständnis, das `hWnd` ist eine Zugriffsnummer, eine Art ID mit deren Hilfe so allerlei angestellt werden kann. Ergänzend sei nochmals erwähnt, daß `hWnd` eigentlich vom Typ `unsignedint` ist, aber auf Grund des nicht Vorhandenseins eines solchen in VB, als `Long`-Wert behandelt wird.

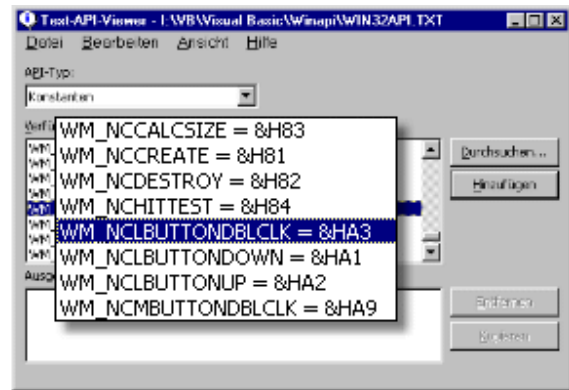
Haben wir diese Voraussetzung akzeptiert, stellt sich die Frage, wozu das ganze? Zwecks Klärung greifen wir den oben angeführten Klick wieder auf. Wir wollen auf Grund der Überschaubarkeit vorläufig annehmen, es gäbe unter Windows nur Klickereignisse und nichts anderes. Die Maus wird bewegt, mal hier hin und mal dort hin. Dabei ist dem Betriebssystem stets bekannt über welchem Fenster, und damit auch dessen `hWnd`, sich der Zeiger gerade befindet. Wird nun z.B. die linke Maustaste betätigt, sendet Windows eine Nachricht an die Nachrichtenprozedur des Fensters über dem das Ereignis gerade stattgefunden hat.

Dort kann an Hand der Nachricht erkannt werden, daß es sich um eine `MouseDown`-Ereignis handelt, um anschließend die entsprechenden, gewollten Schritte zu unternehmen. Das gleiche geschieht, wenn die Taste wieder gelöst wird. Die Nachrichtenfunktion kann die verschiedenen Nachrichten deshalb unterscheiden, da jeder ein fester, genormter `Long`-Wert zugewiesen ist. Für das zu erwartende `MouseDown`-Ereignis der linken Maus-Taste ist dies der Wert = 513 oder in hexadezimaler Darstellung = `&H201`

Insgesamt verfügt Windows über ca. 1000 solcher vordefinierter Nachrichten, auch `Messages` genannt. So viele Zahlen sind schwerlich zu merken. Daher wurden Konstanten mit aussagekräftigen Namen eingeführt. Für unser kleines Beispiel wären das die Konstanten `WM_LBUTTONDOWN` [für `WindowMessage_LeftButtonDown`] und `WM_LBUTTONUP` [für `WindowMessage_LeftButtonUp`]. Weitere sind z.B. dem API-Viewer zu entnehmen. Öffnen Sie ihn und schauen Sie einfach mal nach Einträgen mit dem Präfix `WM_`

Darüber hinaus gibt es eine große Anzahl weiterer controlspezifischer Nachrichten. Der VB-eigene API-Viewer führt lediglich die Geläufigsten. Werte für selten verwendete Messages sind zur Not den C-Headern des Visual-Studios zu entnehmen. [siehe auch Kapitel "Information ist alles"]

Fassen wir kurz zusammen: Windows meint ein Ereignis stünde an, codiert dieses entsprechend einer genormten Konstanten, ruft die Nachrichten-Funktion des betroffenen Fensters auf und übergibt an diese den Wert. Dort wird dieser interpretiert und den Anforderungen gemäß abgearbeitet. In VB entspräche das dann dem Auslösen eines der bekannten Ereignisse, wie `Picture1_Click()` oder `Text1_KeyDown` etc. Der Vorgang ist in VB transparent, daß heißt unter normalen Gegebenheiten nicht erkennbar.



Irgendwann einmal kommt wahrscheinlich jeder an den Punkt, ein standardmäßig nicht implementiertes Ereignis zu benötigen. Wohlwissentlich, daß andere, bestehende Anwendungen mit den Gegebenheiten klarkommen, fragt man sich, wieso die Entwickler von VB gerade bei dem jetzt so dringend erforderlichen geschlumpt haben und verdammt die vermeintlich beschränkte Programmiersprache in alle Himmelsrichtungen.

Weit gefehlt. Ein unter VB nicht gestelltes Ereignis, muß nicht zwangsläufig unabgreifbar sein. Die Betrachtung ist eher in umgekehrter Richtung vorzunehmen. VB bietet mit den bekannten Events lediglich den Luxus, gängige Ereignisse zu kapseln, so daß wir von den ganzen, damit verbundenen Unannehmlichkeiten verschont bleiben. Das spricht uns aber unlängst nicht von der Notwendigkeit frei, uns mit dem Rest der Welt auseinander setzen zu müssen. Hier liegt nur einer der entschiedenen Vorteile der Sprache VB, nämlich der Klassifizierung dieser Prozeduren. In C++ z.B. [ohne MCF] wäre für die Bereitstellung der VB-gegebenen Formereignisse eine Menge Code erforderlich. Wir hingegen beschränken uns gewohntermaßen darauf, mit ein paar lässigen Bewegungen ein Formular hinzuzufügen und zu bestücken.

Aber weiter. Um nicht gegebene Ereignisse müssen wir uns dann halt selber kümmern. Tja, aber wie? Kommen wir zum Joker: Die Fenster-Funktion ist letztendlich eine prozedurale Funktion die irgendwo an einer festen Adresse im Speicher steht. Windows gestattet es ihre Einsprungsadresse (fast) beliebig zu ändern. Was spricht also dagegen, daß wir uns diesen Umstand zu nutze machen und den Zeiger der standardmäßigen VB-Fenster-Funktion auf eine andere, z.B. unsere eigene biegen? Seit VB5 rein gar nichts. Der `AddressOf` Operator [siehe auch Kapitel "Callbacks"] gestattet es die Speicher-Adresse einer beliebigen eigenen Funktion an das API zu übergeben.

Zusammenfassung: Es gibt in VB eine bereits bestehende Fenster-Funktion die sich der Abarbeitung der von Windows gesendeten Nachrichten annimmt. An diese kommen wir aber nicht heran, also verbiegen wir die Einsprungsadresse dieser Funktion auf unsere eigene. Weil wir uns aber nur für ein bis ein paar wenige Nachrichten interessieren und nicht wie in C++ das gesamte Eventhandling selber erledigen wollen, rufen wir nach getaner Arbeit die originale Funktion wieder auf. Praktisch gesehen haben wir uns mit diesem Kniff einfach vor die gängige Fenster-Prozedur gestellt und erhalten sämtliche Nachrichten ab sofort zeitlich vor ihr. Das versetzt uns wiederum in die Lage Nachrichten gezielt zu filtern, zu manipulieren oder zu unterdrücken. Diesen Vorgang nennt man im allgemeinen Subklassifizierung oder Subclassing. Mehr ist nicht dabei.

Schauen wir uns das einfachstmögliche Gerüst für das oben Besprochene einmal an:

```

Const GWL_WNDPROC = (-4&)

Dim PrevWndProc&

Public Sub Init(hWnd As Long)
    PrevWndProc = SetWindowLong(hWnd, GWL_WNDPROC, _
        AddressOf SubWndProc)
End Sub

Public Sub Terminate(hWnd As Long)
    Call SetWindowLong(hWnd, GWL_WNDPROC, PrevWndProc)
End Sub

Private Function SubWndProc(ByVal hWnd As Long, _
    ByVal Msg As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Long) As Long

    SubWndProc = CallWindowProc(PrevWndProc, hWnd, Msg, _
        wParam, lParam)
End Function

```

In der Sub `Init` wird durch Aufruf der Funktion `SetWindowLong` die neue Adresse der Nachrichtenfunktion `SubWndProc` übergeben. Rückgabewert ist hier die Adresse der ursprünglichen VB-Fensterfunktion. `SetWindowLong` ist vielleicht mehr bekannt im Zusammenhang mit den Fensterstilen, Die Konstante `GWL_WNDPROC` bedingt hier aber das Ändern der Einsprungadresse.

Somit ist das Subclassing bereits initialisiert. Um es zu beenden müssen wir den Zeiger wieder auf die ursprüngliche Adresse zurücksetzen. Da wir ihren Wert ja in der Variablen `PrevWndProc` gespeichert haben, ist dies ein leichtes und durch einen erneuten Aufruf von `SetWindowLong` zu erreichen. So geschehen in der Sub `Terminate`.

Hier interessiert uns der Rückgabewert, der ja dann die Adresse von `SubWndProc` enthalten würde, nicht mehr, daher reicht ein einfacher Call.

Achtung: das Subclassing muß unbedingt vor Beendigung des eigenen Programms aufgelöst werden, da die Nachrichten sonst an eine nicht mehr existierende Prozedur geschickt würden und das unweigerlich einen Absturz zur Folge hätte.

Hier noch eine kleine Ergänzung um die Verwirrung etwas zu steigern. Es spricht theoretisch nichts dagegen, ein bereits "subgeclasstes" Fenster wiederum zu subclassen. Man kann sich das ganze als eine Art Kette von Funktionen vorstellen, wobei die erste nach Beendigung die zweite aufruft, diese nach getaner Arbeit die dritte usw.

Im Prinzip gehen wir genauso vor, wenn wir ein VB-Fenster subclassen. Es ergibt sich eine Kette mit zwei Gliedern. Das erste ist unsere eben geschaffene eigene Fensterprozedur, das zweite, die von VB standardmäßig gestellte. Nun ist es leicht nachvollziehbar, daß je länger eine solche Kette ist, desto langsamer wird das gesamte System. Das gilt es zu beachten, und als Folge müßte klar sein, daß Subclassing nur angewandt werden sollte, wenn ein Problem nicht auf einem anderem Wege lösbar erscheint.

Kommen wir nun zum wesentlichen, zu der Funktion "SubWndProc". Hierbei handelt es sich um unsere vor die eigentliche Fensterfunktion geschobene Prozedur. Windows ruft nun bei jedem eintrudelnden Ereignis unsere Funktion auf und übergibt die anliegenden Nachrichten. Eintrudeln ist hierbei stark

untertrieben, es können unter Umständen einige hundert pro Sekunde sein, dies sollte nicht unterschätzt werden. Schauen wir uns also die bei Aufruf übergebenen Parameter etwas genauer an:

hWnd ist relativ unkompliziert und beinhaltet einfach nur den hWnd auf den sich die übergebene Nachricht bezieht. Wenn wir also `Form1` subclassen, wäre dieser Wert gleich `Form1.hwnd`

Msg ist die eigentliche Nachricht und entspricht in der Regel irgendeiner der oben besprochenen vordefinierten Nachrichten. Damit meine ich, es können auch andere sein, da es möglich ist eigene Nachrichten, sogenannte User-Messages, systemweit zu definieren.

wParam und **lParam** übergeben die zur Nachricht gehörende Parameter. Ich weiß das ist sehr allgemein, allerdings gibt es hier keine festen Regularien, vielmehr ist ihre Konstellation vom Nachrichtentyp abhängig und im Zweifelsfalle nachzuschlagen. Bei unserer Eingangs erwähnten Nachricht `WM_LBUTTONDOWN` stehen in `lParam` z.B. die aktuellen Mauskoordinaten, wobei `wParam` den Status spezieller Tasten, wie z.B. `Shift` und `Strg` näher bezeichnet [Bemerken Sie bitte hier die große Ähnlichkeit zu dem Äquivalent-Ereignis in VB]. Im Zusammenhang mit `wParam` und `lParam` gibt es einige Besonderheiten zu beachten, auf die ich später noch zu sprechen kommen werden.

Das waren die Eingangsparameter, als nächstes würden dann unsere zukünftigen Lauscheilen folgen, die wir aber fürs erste außen vor lassen. Abgeschlossen wird unsere Fensterprozedur, indem wir mit `CallWindowProc` die ursprüngliche Fenster-Funktion aufrufen und ihr alle von Windows exklusiv erhaltenen Werte weitergeben. Da es sich um eine Funktion handelt wird im Zweifelsfalle auch ein Rückgabewert erwartet. Hier reichen wir den von der VB-eigenen zurückerstatteten Wert mit `SubWndProc = CallWindowProc(...)` einfach durch.

Kommen wir damit auf eine weitere, wichtige Angelegenheit, die ich bisher verheimlicht habe, wie vielleicht bereits bemerkt wurde: Fenster lassen sich ja bekanntermaßen auch mit der API-Funktion **RegisterClass** erstellen. Mit den entsprechenden Parameter versehen, kreiert Windows nach Ihren Wünschen das passende Fenster. Auffallend ist hierbei, daß keine Adresse einer eventuellen Fensterfunktion übergeben werden kann. Trotzdem ist ein derart generiertes Fenster aber zugänglich für Ereignisse wie Minimieren, Maximieren, verschieben etc. Wie das?

Die Lösung ist recht banal. Windows stellt diese Standardmethoden in Form der Funktion `DefWindowProc` selbst zur Verfügung. Ein mit **RegisterClass** erstelltes Fenster handelt seine Events solange über diese interne Funktion, bis deren Einsprungadresse auf eine eigene Prozedur verbogen wird. Das bedeutet für uns, daß wir, falls wir ein Fenster mit **RegisterClass** erstellen, nach Abschluß unserer Fensterprozedur nicht die Folgeprozedur aufrufen [geht ja auch nicht, da nicht existent], sondern die alles abschließende `DefWindowProc`, die übrigens auch von der VB-eigenen Prozedur nach getaner Arbeit bemüht wird.

Das verleitet uns zu einem weiteren Gedanken. Was wäre, wenn wir in unserer eigenen Prozedur abschließend nicht wie üblich die VB-Fenster-Funktion aufrufen, sondern ebenfalls direkt an `DefWindowProc` weiterleiten? Nunja, eigentlich nichts mehr, im wahrsten Sinne des Wortes. Zumindest nicht in VB, da ja jetzt sämtliche Events an der VB-Fensterprozedur vorbeilaufen. Probieren Sie dies ruhig einmal aus, ich selbst habe es noch nicht getestet.

Ergänzend, aber von großer Bedeutung, sei noch bemerkt, daß eigene Fensterprozeduren sich ausschließlich nur in Standardmodulen aufhalten dürfen, nicht in Formularen und auch nicht direkt in Klassen. Letzteres begründet sich darauf, daß VB mit dem `AddressOf`-Operator nur Module erfassen kann. Es handelt sich hier also um ein VB-typisches, hausgemachtes Manko. Daher setzen wir oben beschriebene Code in ein neues Standardmodul und rufen die `Init`- und `Terminate` Sub wie folgt von einem Formular aus auf:

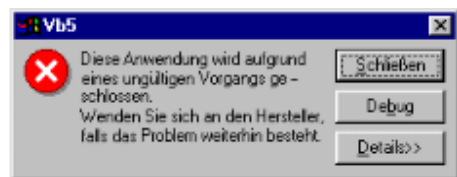
```
Option Explicit
```

```
Private Sub Form_Load()  
    Call Init(Command1.hWnd)  
End Sub
```

```
Private Sub Form_Unload(Cancel As Integer)  
    Call Terminate(Command1.hWnd)  
End Sub
```

Das ist sehr wenig, muß aber so sein, da Standardmodule nicht wie Formulare und Klassen über Initialisierungs- bzw. Terminate-Ereignisse verfügen.

Bevor wir nun zu unserer ersten verwertbaren Subklassifizierung kommen, noch ein paar wichtige Hinweise. Es gibt hier einige Besonderheiten im Zusammenhang mit der IDE und der Strukturierung zu berücksichtigen.



So sollten Sie die folgenden Erläuterungen aufmerksam lesen und gegebenenfalls alles einmal praktisch durchspielen, um sich mit den Ecken und Kanten der Methoden vertraut zu machen:

Allgemeine Sicherheitshinweise

- Wenn Sie an Ihren Subclassings entwickeln, ist es unumgänglich, daß Ihnen hin und wieder die IDE inklusive des zu bearbeitenden Programms wegen unzulässiger Vorgänge geschlossen wird. Das ist nicht weiter schlimm und gehört hier zum Handwerk, heißt aber in der Konsequenz, daß Sie Ihr Programm grundsätzlich vor dem Starten sichern sollten, sonst ist der Ärger später groß.

Fehler & Debuggen

- Beides geht einher. Treten Fehler auf oder werden Haltepunkte gesetzt, so wird wie gewohnt im Entwurfsmodus Mr. Debugger angeworfen. Der verharrt in betroffener Zeile solange bis ihm aufgetragen wird fortzufahren. Das würde er auch tun, wenn das aktuelle Programm für Events weiterhin empfänglich wäre. Dummerweise können wir uns aber gerade inmitten in der Routine befinden, durch die das benötigte Ereignis für die Fortsetzung ginge. Dieses kann aber erst dann abgearbeitet werden, wenn der aktuelle Durchlauf vollzogen ist und dies geht wiederum nur, wenn wir den Haltemodus des Debuggers ausschalten. Also eine Sackgasse. Bei Haltepunkten gibt es unter Umständen nur ein Entkommen, nämlich über die Tastenkombination `Strg + Umschalt + F9`. Fehlermeldungen können mit der `On Error Resume Next` Anweisung unterdrückt werden. Andernfalls hilft manchmal nur der Affengriff [`Strg + Alt + Entf`] um die Anwendung zu schließen

Verschachtelungen als unerwünschte Rekursion

- Ein Überlauf des Stapel kann eintreten, wenn aus einer Fensterprozedur ein Vorgang ausgelöst wird, welcher eine Nachricht an die Fensterprozedur veranlaßt, diese Nachricht wiederum löst dann erneut den ursprünglichen Vorgang aus der daraufhin ... usw. solange, bis der Stack überläuft. Daraus folgt letztendlich eine nicht abfangbare Fehlermeldung und damit ein Absturz. So etwas kann umgangen werden, indem durch das Setzen von statischen Flags ein weiteres Auslösen verhindert wird, so daß das ordnungsgemäße Abarbeiten der aktuellen Nachricht gewährleistet ist.

Das große Beenden

- Ein immer wieder gemachter und Verwunderung auslösender Fehler, ist das Schließen des Programms bei laufendem Subclassing über den Stop-Button der IDE. Durch dessen abruptes Beenden wird im Formular nicht das übliche `Form_Unload` Ereignis ausgelöst, so daß das Subclassing unterminiert bleibt. Es kommt zum Absturz. Abhilfe schafft das Terminieren der Nachrichtenfunktion mittels Abfangen der `WM_DESTROY` Nachricht oder das grundsätzliche Schließen des Programms über die X-Schaltfläche in

der Titelleiste oder durch einen separaten Button. Wohlhermerkt, dieser Effekt tritt nur in der IDE auf und kann daher in der kompilierten Exe unberücksichtigt bleiben.

Nun, jetzt wären wir soweit unser Wissen an anhand eines kleine Beispiels unter Beweis zu stellen. Es sollte etwas halbwegs nützlich, aber als Ereignis in VB noch nicht vorliegendes sein. Vorschlag zur Güte, ein MouseMove-Ereignis für den NonClient-Bereich. Das ist die Fläche des Formulars, die normalerweise in VB nicht direkt erreichbar ist. Genauer gesagt die Menüleiste, die `Caption` als auch je nach Einstellung die 4 Pixel breiten Ränder. Sie erkennen diese Fläche in dem zugehörigen Beispiel sehr leicht, da sie einen farblichen Kontrast zur Client-Area bietet. Schauen wir einmal in den API-Viewer, ob wir einen passenden Nachrichtentyp zu unserer Aufgabe finden. Wir stoßen unter anderem auf folgende drei Konstanten:

```
Const WM_NCLBUTTONDBLCLK = &HA3
Const WM_NCLBUTTONDOWN = &HA1
Const WM_NCLBUTTONUP = &HA2
```

Das sieht schon sehr vielversprechend aus. Jetzt müssen wir nur in unserer Fensterprozedur auf diese Nachrichten lauern. Das machen wir vorerst mit nur einer Message wie folgt:

Option Explicit

```
Private Declare Function SetWindowLong Lib "user32" _
    Alias "SetWindowLongA" (ByVal hWnd As Long, _
    ByVal nIndex As Long, ByVal dwNewLong As Long) _
    As Long
```

```
Private Declare Function CallWindowProc Lib "user32" _
    Alias "CallWindowProcA" (ByVal lpPrevWndFunc _
    As Long, ByVal hWnd As Long, ByVal Msg As _
    Long, ByVal wParam As Long, ByVal lParam As _
    Long) As Long
```

```
Const GWL_WNDPROC = (-4&)
```

```
Dim PrevWndProc&
```

```
Public Sub Init(hWnd As Long)
    PrevWndProc = SetWindowLong(hWnd, GWL_WNDPROC, _
        AddressOf SubWndProc)
End Sub
```

```
Public Sub Terminate(hWnd As Long)
    Call SetWindowLong(hWnd, GWL_WNDPROC, PrevWndProc)
End Sub
```

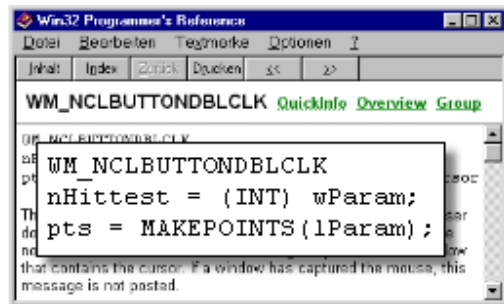
```
Private Function SubWndProc(ByVal hWnd As Long, _
    ByVal Msg As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Long) As Long
```

```
    If Msg = WM_NCLBUTTONDBLCLK Then MsgBox ("DoubleClick")
```

```
    SubWndProc = CallWindowProc(PrevWndProc, hWnd, Msg, _
        wParam, lParam)
```

```
End Function
```

Zum Testen Doppel-Klicken Sie bitte auf eine beliebige Stelle des NonClient-Bereichs, wie z.B. die Menüleiste, die Caption oder die Ränder. Er ist leicht zu erkennen da der Client Bereich sich durch seine weiße Hintergrundfarbe abhebt. Bisher haben wir nur die Nachricht ausgewertet, was aber bieten dabei die übergebenen Paramter wParam und lParam? Ein Blick in die [Win32.hlp](#) verrät, daß wParam das Ergebnis des sogenannten HitTests wiedergibt, welcher Aufschluß darüber gibt, auf was letztendlich der Klick ausgeübt wurde, hierzu später mehr. lParam beinhaltet die absoluten x-, y-Koordinate, der Stelle auf die geklickt wurde. Wohlgermerkt die absoluten Bildschirmkoordinaten. Um die relativen zu erhalten, müssen die absoluten Eckkoordinaten des Fenster hiervon abgezogen werden.



Fein, das wollen wir genauer wissen. Doch gibt es noch einen kleinen Stolperstein: Die übergebene Long-Variable bietet zwei Werte vom Typ `short` in einem einzigen, wobei das obere Word die y-Koordinate und das untere den x-Wert darstellt. Da VB keine Variable vom Typ `short` kennt, müssen wir hier ein wenig tricksen und uns eine kleine Funktion zwecks Umrechnung schreiben:

```
Private Sub GetShort(Value&, Lo&, Hi&)
    Lo = Value And &H7FFF
    Hi = Value \ &H10000
End Sub
```

Weiterhin interessiert uns die Sache mit dem HitTest in wParam. Ein erneuter Blick in den API-Viewer verrät uns weiterhin folgende Konstanten:

```
Const HTBORDER = 18
Const HTBOTTOM = 15
Const HTBOTTOMLEFT = 16
'...
'etc.
```

Da aber reine Zahlen als Rückgabewert nicht sehr aussagekräftig sind, stricken wir uns eine zweite, einfache Hilfsfunktion, die aus dem von wParam zurückgegebenen Wert einen lesbaren Text erstellt:

```
Private Function HitTestToString(HitTest As Long) As String
    Dim aa As String

    Select Case HitTest
        Case HTBORDER: aa = "HTBORDER"
        Case HTBOTTOM: aa = "HTBOTTOM"
        Case HTBOTTOMLEFT: aa = "HTBOTTOMLEFT"
        Case HTBOTTOMRIGHT: aa = "HTBOTTOMRIGHT"
        Case HTCAPTION: aa = "HTCAPTION"
        Case HTCLIENT: aa = "HTCLIENT"
        Case HTERROR: aa = "HTERROR"
        Case HTGROWBOX: aa = "HTGROWBOX"
        Case HTHSCROLL: aa = "HTHSCROLL"
        Case HTLEFT: aa = "HTLEFT"
        Case HTMAXBUTTON: aa = "HTMAXBUTTON"
        Case HTMENU: aa = "HTMENU"
        Case HTMINBUTTON: aa = "HTMINBUTTON"
        Case HTNOWHERE: aa = "HTNOWHERE"
        Case HTRIGHT: aa = "HTRIGHT"
        Case HTSYSTEMMENU: aa = "HTSYSTEMMENU"
```

```

Case HTTOP:          aa = "HTTOP"
Case HTTOPLEFT:     aa = "HTTOPLEFT"
Case HTTOPRIGHT:    aa = "HTTOPRIGHT"
Case HTTRANSPARENT: aa = "HTTRANSPARENT"
Case HTVSCROLL:     aa = "HTVSCROLL"
Case HTCLOSE:       aa = "HTCLOSE"
Case HTHELP:        aa = "HTHELP"
Case Else:          aa = CStr(HitTest)
End Select

```

```

HitTestToString = aa
End Function

```

Zudem soll neben dem Click- auch ein MouseMove-Event ausgelöst werden. Dazu bedienen wir uns der Nachricht WM_NCMOUSEMOVE die sich ansonsten in nichts von den bereits verwendeten Maus-Tastennachrichten unterscheidet.

Um die Ereignisse letztendlich auch im Formular nutzen zu können, fügen wir in Form1 neben einigen Labeln zur Anzeige, folgenden Code ein:

```

Public Sub NonClient_LeftMouseButton(State As Integer, _
                                       x As Long, _
                                       y As Long)

Label4.Caption = x
Label5.Caption = y

Select Case State
Case 1: Label6.Caption = "MouseDown"
Case 2: Label6.Caption = "MouseUp"
End Select
End Sub

```

```

Public Sub NonClient_LeftDbClick(x As Long, y As Long)
Static DblClkCount&

Label4.Caption = x
Label5.Caption = y

DblClkCount = DblClkCount + 1
Label7.Caption = DblClkCount
End Sub

```

Die Subs müssen vom Typ `Public` sein, damit sie vom Modul aus aufgerufen werden können. Das Modul als solches wird jetzt noch um die neue Nachricht und die Auswertfunktionen erweitert, womit unser kleines Projekt auch schon abgeschlossen wäre:

```

Private Function SubWndProc(ByVal hWnd As Long, _
                             ByVal Msg As Long, _
                             ByVal wParam As Long, _
                             ByVal lParam As Long) As Long

Dim x&, y&, Text$, State%

Select Case Msg
Case WM_NCLBUTTONDOWN, _
     WM_NCLBUTTONUP, _
     WM_NCLBUTTONDBLCLK:

```

```
Call GetShort(lParam, x, y)
x = x - Form1.Left \ Screen.TwipsPerPixelX
y = y - Form1.Top \ Screen.TwipsPerPixelY

If Msg = WM_NCLBUTTONDOWN Then
    State = 1
ElseIf Msg = WM_NCLBUTTONUP Then
    State = 2
End If

If Msg = WM_NCLBUTTONDBLCLK Then
    Call Form1.NonClient_LeftDbClick(x, y)
Else
    Call Form1.NonClient_LeftMouseButton(State, x, y)
End If

Case WM_NCMOUSEMOVE:

    Call GetShort(lParam, x, y)
    x = x - Form1.Left \ Screen.TwipsPerPixelX
    y = y - Form1.Top \ Screen.TwipsPerPixelY
    Text = HitTestToString(wParam)
    Call Form1.NonClient_MouseMove(Text, x, y)

End Select

SubWndProc = CallWindowProc(PrevWndProc, hWnd, _
                             Msg, wParam, lParam)

End Function
```

Sicherlich ist das hiesige Kapitel nur eine knappe Einführung zu diesem umfassenden Thema. Letztendlich kann man hier sehr schnell abschweifen und halbe Bücher füllen, da es im wesentlichen um die unmaskierte Funktionsweise von Windows geht. Wie bereits Eingangs erwähnt, gibt es eine Fülle von Nachrichten, mit noch mehr unterschiedlichen Zusammenhängen, in denen sie in Erscheinung treten können.

Um ein Beispiel zu geben: Mit `lParam` können auch Strukturen erhalten werden, allerdings als Zeiger, da VB keine Zeiger beherrscht, muß sich hier mit **CopyMemory** beholfen werden. Wir erinnern uns im Kapitel "Strukturen respektive Typen" diese Problem gelöst zu haben, indem wir einen auf diese Weise referenzierten Typen in eine Platzhalterstruktur kopierten, an diesem die gewünschten Änderungen vornahmen und ihn abschließend zurückkopierten, damit die Manipulationen auch wirksam wurden.

Abschließend möchte ich noch bemerken, daß Subclassing zwangsläufig einen Einblick in Windows und dessen vielseitiges Nachrichtenwesen gewährt und damit auch wesentlich zum besseren Verständnis der Abläufe unter VB beiträgt.

Schlußwort

Wahrscheinlich könnte man noch über Tage hinweg in dieser Art weiter oberlehren, ohne dabei an thematische Grenzen zu stoßen. Es gibt selbst für die Programmiersprache C mehrbändige Kompendien über das API und seine in ihm wohnenden Funktionen und Eigenarten. Ich hoffe mit dieser Lektüre Ihre Neugierde auf die Mächtigkeit dieser Windows-Schnittstelle weiter angefacht und Sie dabei gleichzeitig, durch das Anbieten einer Reihe von grundsätzlichen Methodiken, für den alltäglichen Einsatz des APIs gerüstet zu haben.

Verzagen Sie nicht, die meisten Nüsse im Umgang mit dem API unter VB sind zu knacken, auch wenn beim Lösen eines anfänglich unscheinbaren Problems, schon einmal ein ganzer Tag vergehen kann. Eine gute Recherche zu Beginn, spart oftmals späteres stundenlanges Fluchen. Hin und wieder ist auch einfach nur sture Hartnäckigkeit der Schlüssel zum Ziel.

Da ich stetig bemüht bin, den vorliegenden Artikel abzurunden und auch zu erweitern, würde ich mich freuen, wenn Sie mir mitteilten, falls Ihnen ein Thema unverständlich oder sogar falsch erschien. Verbesserungsvorschläge sind eh grundsätzlich erwünscht. Auch die vielen kleinen Rechtschreib- und grammatikalischen Fehler dürfen Sie gerne unter Nennung Ihrer Version [s. Deckblatt] und der jeweiligen Seite plus Zeile per Email antadeln unter: api.handbuch@activevb.de

Götz Reinecke