Win32API-Tutorials für Delphi

von Michael Puff

PDF Version 2.2 [2006-02-04]

Inhaltsverzeichnis

Vorwort

Wie alles begann

Angefangen habe ich, wie wohl jeder Delphi-Programmierer, mit der VCL. Doch irgendwann kommt man an den Punkt, an dem man neue Herausforderungen sucht ... und im günstigsten Fall auch findet. Ich fand meine in der Delphi-Programmierung ohne die VCL, also rein auf Win32-API-Basis. Ich darf sagen, dass es sich mit der Zeit zu meinem Spezialgebiet entwickelt hat.

Begonnen hat alles mit einem Tutorial von Assarbad. Na ja, eigentlich war es nur eine Einführung, gespickt mit Hintergrundwissen darüber, wie das ganze eigentlich funktioniert. Doch es hatte mich gepackt, und nachdem ich die ersten wackeligen Schritte unternommen hatte und anfing, mir eigenes Wissen diesbezüglich anzueignen, entschied ich mich, es an andere weiterzugeben. Daraus sind dann "Luckies nonVCL-Tutorials" in ihrer ersten Version entstanden. Diese bilden auch heute noch den Kern und die Basis für die Tutorials in ihrer jetzigen Form. Angeboten hatte ich sie zum Lesen online auf meiner Homepage und zum Download in einem Zip-Archiv (HTML-Seiten plus Sourcen der Demos).

Versionschaos?

Ein befreundeter Programmierer, Mathias Simmack, sprach mich dann darauf an, die Tutorials in einer anwenderfreundlicheren Version zu präsentieren. Und die Idee, sie in Form einer CHM-Hilfe zu veröffentlichen, ward geboren. Nach wohl einigen Wochen mühseliger Arbeit konnte die CHM-Version dann das Licht der Öffentlichkeit erblicken. Verpackt in einem benutzerfreundlichen Setup, für das sich auch Mathias verantwortlich zeigte. Im Laufe der Zeit hat er sich dadurch zu einem Experten für Hilfedateien und Setups entwickelt. Und noch heute ist er verantwortlich für die CHM-Version und das Setup. Außerdem steuerte er dann auch noch einige Tutorials bei.

Nach einiger Zeit wurden dann vereinzelt Stimmen nach einer besser druckbaren Version laut. Und da auch ich mal etwas Handfestes in der Hand haben wollte, sprich: Schwarz auf Weiß, machte ich mich ans Werk und bereitete die Tutorials als PDF-Datei auf. Als Grundlage dienten mir die aufbereiteten HTML-Dateien der CHM-Version. Somit ist die PDF-Version inhaltlich mit der CHM-Version identisch. Diese Konsistenz soll auch in Zukunft gewahrt bleiben. Es gab und es wird immer nur die "Win32-API-Tutorials für Delphi" geben - wenn auch in unterschiedlichen Formaten. Damit sollte für jeden Geschmack etwas dabei sein.

Pläne für die Zukunft

Ich hoffe, dass die Tutorials nicht nur in der Delphi-Praxis, meinem Stamm-Forum, und im Delphi-Forum Verbreitung finden, sondern mit der Zeit auch in anderen Delphi-Communities auf größeres Interesse stoßen. Ich hoffe, dass die PDF-Version dazu beiträgt diese Ziele zu erreichen. Ein weiteres Ziel wäre es, die Tutorials auf einer Zeitschrift oder einem großen Download-Portal für Software oder eBooks unterzubringen.

Und dann wäre da noch der Traum, welcher aber erstmal auch nur ein Wunschtraum bleiben wird: eine Veröffentlichung als Buch.

Danksagung

An erster Stelle steht da natürlich Assarbad aka Oliver, mit dessen Tutorial alles angefangen hat, und der mir in der Anfangszeit per ICQ immer mit Rat und Tat beiseite gestanden hat. Als nächstes wäre dann da noch Nico Bendlin zu nennen, dessen Übersetzungen der "Iczelion's Win32 ASM Tutorials" nach Delphi als Grundlage für das ein oder andere Tutorial gedient haben. Nicht zu vergessen natürlich Mathias, der mit der CHM-Version und den dazugehörigen Setups eine hervorragende Arbeit geleistet hat und mir auch als Autor, Kritiker und Berater immer zur Seite stand und steht.

In diesem Sinne wünsche ich viel Spaß beim Lesen der "Win32-API-Tutorials für Delphi".

Euer Michael Puff, Vellmar im September 2004

nonVCL, was ist das eigentlich?

Unter der nonVCL-Programmierung versteht man die Programmierung ohne die Verwendung der VCL. Es gibt keine Komponenten, keine Formulare in dem Sinne ... usw. Stattdessen greift man direkt auf das API (Application Programmers Interface) zurück. Wenn Sie bereits Erfahrung mit der VCL-Programmierung haben, dann wird Ihnen zuerst klar werden (müssen), dass Sie sich bei reinen API-Programmen um viele Dinge selbst kümmern müssen. Der Nachteil dabei ist, dass die Quelltexte solcher Projekte u.U. etwas umfangreicher und mitunter auch unübersichtlicher werden. Andererseits haben die kompilierten Exe-Dateien nur einen Bruchteil der Größe der VCL-Versionen. Und das ist für viele Programmierer Grund genug, kleinere Tools komplett ohne VCL zu entwickeln.

Allerdings dürfen Sie die Vorteile der VCL nicht vergessen. Unserer Meinung nach ist es nicht vertretbar, ein umfangreiches Projekt mit vier, fünf oder mehr Formularen ausschließlich auf API-Basis zu entwickeln. Sicher wird sich das realisieren lassen, aber hier sind Sie mit der VCL eindeutig schneller, und Sie können bequemer arbeiten.
Ideal ist die API-Programmierung bei der Entwicklung von kleineren Projekten und Tools, die vielleicht nur ein oder zwei Fenster oder Dialoge haben und bei denen der Funktionsumfang nicht so gewaltig ist.

Im Gegensatz zum herkömmlichen Weg müssen Sie bei nonVCL-Programmen generell für eins sorgen: Ihr Programm muss aktiv bleiben. Deshalb besteht ein typisches nonVCL-Grundgerüst aus drei Teilen -

- der Hauptfunktion (in PASCAL durch **begin** und **end.** gekennzeichnet)
- der Nachrichtenschleife
- der Nachrichtenfunktion

Beispiel:

```
// Nachrichtenfunktion
function WndProc(wnd: HWND; uMsg: UINT; wp: WPARAM; lp: LPARAM):
  LRESULT; stdcall;
begin
  Result := 0;

  case uMsg of
    WM_CREATE:
      // Funktionen ausführen
    WM_DESTROY:
      PostQuitMessage(0);
    else
      Result := DefWindowProc(wnd,uMsg,wp,lp);
  end;
end;

// Hauptfunktion
var
  msg : TMsg;
begin
  // Fensterklasse registrieren, & Fenster erzeugen
  { ... }

  // Nachrichtenschleife
  while(GetMessage(msg,0,0,0)) do
    begin
      TranslateMessage(msg);
      DispatchMessage(msg);
    end;
end.
```

Bevor Sie jedoch beginnen nonVCL-Programme zu schreiben, sollten Sie sich mit den Grundlagen der Programmierung in PASCAL respektive ObjectPascal (Delphi) auskennen. Es macht nicht viel Sinn, völlig ohne Vorkenntnisse in diese Materie einsteigen zu wollen. Frust dürfte wohl die Folge sein.

Außerdem sind weitergehende API-Dokumentationen unbedingt empfehlenswert. Als erstes wären da die Win32-API-Hilfedateien von Borland zu nennen. Diese liegen Delphi bei, sind allerdings auch veraltet und (in Bezug auf neue Controls) unvollständig. Sofern möglich nutzen Sie also gleich die Quellen von Microsoft, als da wären:

- das Microsoft Developers Network (MSDN) (www.msdn.microsoft.com)
- das Platform SDK (PSDK) (http://www.microsoft.com/msdownload/platformsdk/sdkupdate)

Die Tutorials bauen weitgehend aufeinander auf. Wenn Sie also nicht gezielt ein bestimmtes Thema suchen, dann ist es ganz sinnvoll, sie der Reihe nach durchzuarbeiten.

Informationen

Verfügbare Versionen / zugehöriges Material

- chm-Version mit Setup
- Sourcen der Demos mit Setup
- PDF-Version
- Sourcen der Demos in einem Zip-Archiv

Bezugsquellen / Download-Möglichkeiten

www.luckie-online.de
www.simmack.de

Die Autoren

Michael Puff	Mathias Simmack	Thomas Liebetraut
Fenster	Suchen- / Ersetzendialog	Ressourcenskripte erstellen
Textsausgabe in Labels	Tooltips / Hints	
Schaltflächen	Das IP-Adress-Eingabefeld	
Checkboxen und Radiobuttons	Der Treeview	
Arbeiten mit Eingabefeldern	Die Rebar	
Arbeiten mit Listboxen	Das SysLink-Control	
Arbeiten mit Comboboxen	Verbindung zur TNA	
Ein Menü hinzufügen	Hotkeys und Shortcuts	
Dialoge aus Ressourcen aufrufen	Datum und Uhrzeit	
Dateien öffnen und speichern	Die Verwendung von Ini-Dateien	
Die Schriftart ändern	Die Registry	
Fortschrittsanzeige	Anwendung für die Systemsteuerung	
Die Statuszeile	Einen Assistenten erstellen	
Die Toolbar	HLP-Hilfedateien	
Die Trackbar	CHM-Hilfedateien	
Der Listview		
Tabsheets		
Der Timer ohne die VCL		
Subclassing		
Grundlagen der GDI, Teil 1		

Kontaktmöglichkeiten

Michael Puff:
Homepage : www.luckie-online.de
E-Mail : tutorials@luckie-online.de

Mathias Simmack:
Homepage : www.simmack.de

Thomas Liebetraut:
Homepage : www.tommie-lie.net
E-Mail : liebetraut@web.de

1. Fenster und Controls

1.1. Fenster

1.1.1. Das Fenster erzeugen

Unter Windows werden normalerweise rechteckige Fenster erzeugt, mit denen der Benutzer interagiert. Daher stellen wir erst einmal einige grundlegende Merkmale eines Fensters zusammen, die wir dann in einem Programm definieren wollen:

Wert	Bedeutung
x, y, w, h	linke obere Ecke (x, y), und Breite und Höhe (w, h) des Fensters
Icon	Icon des Fensters
SysMenu	das Systemmenü, das sich beim Klicken auf das Icon bzw. mit ALT-Leertaste öffnet
Rahmen	kein Rahmen, fester Rahmen, bzw, mit der Maus veränderbarer Rahmen
Caption	Beschriftungstext in der Titelzeile
Minimizebox, Maximizebox	Schaltfläche zum Minimieren und Maximieren des Fensters
Cursor	Form des Mauszeigers innerhalb des Fensters
Background	Hintergrundfarbe, -muster des Fensters
Menu	Menü, das zum Fenster gehört

Man sieht: als Programmierer muss man sich allein schon wegen der grafischen Unterstützung von Windows-Programmen um eine Vielzahl grafischer Details kümmern, die mehr mit dem Design als mit dem Programmierziel zu tun haben. Die Fenstermerkmale werden an zwei Stellen definiert. Die erste Hälfte wird im Fensterklassen-Record, die zweite Hälfte wird mit der Funktion "`CreateWindow(Ex)`" festgelegt. Im Demo-Quellcode finden Sie z.B. die Befehlszeile:

```
CreateWindowEx(0,
               ClassName,
               AppName,
               WS_CAPTION or WS_VISIBLE or WS_SYSMENU or WS_MINIMIZEBOX or
               WS_MAXIMIZEBOX or WS_SIZEBOX,
               CW_USEDEFAULT,  // Position von Links
               CW_USEDEFAULT,  // Position von oben
               WindowWidth,    // Breite (hier Konstante)
               WindowHeight,   // Höhe (hier Konstante)
               0,
               0,
               hInstance,
               nil);
```

Experimentieren Sie einfach mit den kommentierten Werten, und beobachten Sie die Ergebnisse. Sie können auch negative Werte eingeben. Wollen Sie sich weder um die Position noch um die Größe des Fensters kümmern, dann verwenden Sie den Konstantenwert `CW_USEDEFAULT` anstelle von Pixelangaben.

Ohne vorgreifen zu wollen - Stellen wir uns mal die Frage: »Woher weiß "`CreateWindowEx`" welche Fensterklasse es als Grundlage nehmen soll?«. In dem Fall ist es recht einfach. Im Record haben wir mit "`wc.lpszClassName := Classname;`" die Möglichkeit, einen Klassennamen anzugeben. In dem Fall eine Konstante aus dem Programmkopf. Den selben Namen übergeben wir als zweiten Parameter an die Funktion "`CreateWindowEx`".

TWndClassEx-Definition

```
typedef struct _WNDCLASSEX {
    UINT    cbSize;         // Größe des Records
    UINT    style;          // Stil
    WNDPROC lpfnWndProc;    // Zeiger auf Nachrichtenfunktion
    int     cbClsExtra;
    int     cbWndExtra;
    HANDLE  hInstance;      // Anwendungsinstanz
    HICON   hIcon;          // Symbol-Handle
    HCURSOR hCursor;        // Cursor-Handle
    HBRUSH  hbrBackground; // Hintergrund der ClientArea
    LPCTSTR lpszMenuName;   // MainMenu-Handle
    LPCTSTR lpszClassName; // Klassenname
    HICON   hIconSm;        // Symbol-Handle (kleines Symbol)
} WNDCLASSEX;
```

1.1.2. Symbol und Mauszeiger festlegen

Wie kommt nun eigentlich das Icon in die Titelzeile des Fensters? Im Beispiel wird das Icon aus einer Ressourcendatei geladen, und zwar an dieser Stelle:

```
wc.hIcon := LoadIcon(hInstance, MAKEINTRESOURCE(100));
```

Voraussetzung dafür ist natürlich, dass diese Ressourcendatei vorher erzeugt und dann auch in den Quellcode eingebunden wurde. Das Symbol selbst wird dann mit der Funktion "`LoadIcon`" geladen, die als Parameter ein Handle der Anwendungsinstanz und den Iconnamen oder die Ressourcen-ID des Symbols erwartet. Wenn Sie eine Ressourcen-ID verwenden, müssen Sie das Makro "`MAKEINTRESOURCE`" benutzen, um den Integerwert in einen Ressourcentyp umzuwandeln.
Wenn Sie auf Ressourcendateien verzichten wollen, können Sie bestimmte Standardsymbole von Windows benutzen. Dazu ändern Sie die obige Zeile wie folgt ab:

```
wc.hIcon := LoadIcon(hInstance, IDI_WINLOGO);
```

Und schon sehen Sie das Windows-Logo als Icon in der Titelzeile. In der Hilfe zu der Funktion "`LoadIcon`" finden Sie weitere Werte, die Sie verwenden können.

Mit dem Mauszeiger lässt sich entsprechend verfahren. Das Beispielprogramm lädt den Standard-Mauszeiger:

```
wc.hCursor := LoadCursor(0, IDC_ARROW);
```

Trotz der heutigen schnellen Rechner keine Seltenheit: `IDC_WAIT`. Sie wissen sicher schon, welcher Cursor das ist? Richtig - die altbekannte Sanduhr. Weitere Cursortypen finden Sie in der Hilfe zur Funktion "`LoadCursor`".

1.1.3. Die Titelzeile

In unserem Beispielprogramm finden wir in der Titelzeile (v.l.n.r): das Systemmenü, den Fenstertitel und die drei Schaltflächen (Minimieren, Maximieren, Schließen):

Im Programmcode sind das diese Angaben:

```
CreateWindowEx(0, ClassName, AppName,
               WS_CAPTION or        // Fenster hat eine Titelzeile
               WS_VISIBLE or        // Fenster ist sichtbar
               WS_SYSMENU or        // Systemmenü ist vorhanden
               WS_MINIMIZEBOX or    // Minimieren-Schaltfläche ist vorhanden
               WS_MAXIMIZEBOX or    // Maximieren-Schaltfläche ist vorhanden
               WS_SIZEBOX,          // Fenstergröße lässt sich ändern
               CW_USEDEFAULT, CW_USEDEFAULT, WindowWidth, WindowHeight,
               0, 0, hInstance, nil);
```

Auch hier sollten Sie ein wenig mit den Angaben experimentieren. Lassen Sie Konstanten weg, fügen Sie andere hinzu und schauen Sie was passiert. Mehr zu den Fensterstilen erfahren Sie in der Hilfe unter "`CreateWindow`" und "`CreateWindowEx`". Beide Funktionen erzeugen übrigens ein Fenster, allerdings besitzt letztere eine erweiterte Struktur und lässt dadurch weitere Stile zu.

1.1.4. Der Anwendungsbereich

Der Anwendungsbereich (VCL = "Clientarea") ist der Teil des Fensters, den Sie für Buttons, Listen ... kurz gesagt: für die von Ihnen gewünschten Elemente benutzen können. Seine Hintergrundfarbe legen Sie im Fensterklassen-Record in dieser Zeile fest:

```
wc: TWndClassEx = (
  ...
  hbrBackground   : COLOR_APPWORKSPACE;
  ...
);
```

In dem Fall wird mit `COLOR_APPWORKSPACE` die Standardfarbe Ihres Systems für 3D-Objekte einstellt. Weitere Konstanten finden Sie in der Hilfe zu `WNDCLASS` oder `WNDCLASSEX`. Reichen Ihnen die vorgegebenen Farben nicht, dann können Sie eigene erstellen. Ergänzen Sie dazu das Beispielprogramm im Hauptteil um die folgende Zeile:

```
begin
  wc.hInstance  := hInstance;
  ...

  // diese Zeile ergänzen ->
  wc.hbrBackground := CreateSolidBrush(RGB(0,150,255));

  ...
end.
```

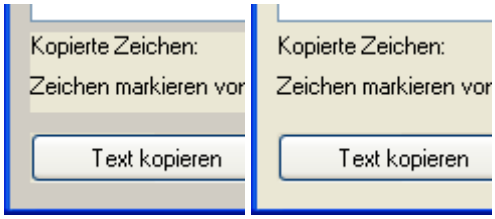
Auf die gleiche Weise können Sie nun mit Hilfe der RGB-Angabe jede beliebige Farben mischen. Oder weisen Sie doch mal "`wc.hbrBackground`" den Wert Null zu ... »... and see what happens ...« ;o)

Hinweis für Windows XP

Wie Ihnen das Bild links zeigt, wird unter Windows XP der Fensterhintergrund evtl. zu dunkel dargestellt. Das liegt an der Verwendung von `COLOR_APPWORKSPACE` (wie eingangs gezeigt) bei aktiven Themes. Als Abhilfe sollten Sie mit der Funktion "`GetSysColorBrush`" die Farbe für 3D-Elemente laden und benutzen:

```
wc.hbrBackground := GetSysColorBrush(COLOR_3DFACE);
```

Wie Sie im Bild rechts sehen können, wird dann der Hintergrund korrekt dargestellt:



Sie können diesen Befehl auch verwenden, wenn Sie keine Themes aktiviert haben bzw. ein anderes Betriebssystem als Windows XP benutzen (9x, ME, NT und 2000). Eine Übersicht über die möglichen Farbwerte finden Sie unter dem Befehl "`GetSysColor`" im PSDK.
Der Unterschied zwischen beiden Befehlen ist, dass "`GetSysColor`**`Brush`**" die gewünschte Farbe gleich als so genannten *Brush* zurückliefert, der sofort mit der Membervariablen `hbrBackground` (s. `TWndClassEx`-Record) benutzt werden kann. Dagegen müssten Sie das Ergebnis von "`GetSysColor`" (ein `dword`-Wert mit den RGB-Informationen der Farbe) erst bspw. mit "`CreateSolidBrush`" umwandeln, um ihn für den Fensterhintergrund nutzen zu können.

1.1.5. Die Nachrichtenfunktion

Zuständig für Nachrichten innerhalb unseres Programms ist eine eigene Funktion. Die meisten Programme verwenden den Namen "`WndProc`" für diese Funktion, der auch so in der Microsoft-Hilfe zu finden ist. Auch unser Beispielprogramm hat in der Zeile

```
wc: TWndClassEx = (
  ...
  lpfnWndProc     : @WndProc;
  ...
);
```

diese Funktion deklariert. Fensternachrichten ("Windows Messages") beginnen üblicherweise mit dem Präfix "`WM_???`". Der beste Weg, diese Nachrichten zu filtern und abzuarbeiten, dürfte eine **case**-Schleife sein. Nachrichten, für die wir keine besondere Aktion vorgesehen haben, übergeben wir an die allgemeine Nachrichtenfunktion von Windows. Das System kümmert sich dann darum. Der folgende Auszug aus dem Beispielprogramm zeigt Ihnen das Filtern der Nachricht "`WM_CREATE`", die beim Erzeugen unseres Fensters aufgerufen wird:

```
function WndProc(hWnd: HWND; uMsg: UINT; wParam: wParam; lParam: LParam):
  lresult; stdcall;
var
  x, y : integer;   //Variablen für Fensterposition
begin
  Result := 0;

  case uMsg of
    WM_CREATE:
      begin
        {Fenster zentrieren}
        x := GetSystemMetrics(SM_CXSCREEN);   //Screenhöhe & -breite
        y := GetSystemMetrics(SM_CYSCREEN);
        {Fenster auf neue Position verschieben}
        MoveWindow(hWnd, (x div 2) - (WindowWidth div 2),
          (y div 2) - (WindowHeight div 2),
          WindowWidth, WindowHeight, true);
      end;
    else
      Result := DefWindowProc(hWnd,uMsg,wParam,lParam);
  end;
end;
```

Ein Wort zu "WM_DESTROY" -
Diese Nachricht wird gesendet, wenn das Fenster im wahrsten Sinn des Wortes zerstört wird. In diesem Fall müssen wir mit "PostQuitMessage(0);" antworten und so die Nachricht "WM_QUIT" erzeugen, die die Nachrichtenschleife beendet. In der Hilfe heißt es dazu (frei übersetzt):

Platform SDK:

Die Nachricht **WM_QUIT** bezeichnet die Aufforderung, eine Anwendung zu beenden und wird erzeugt, wenn die Anwendung die Funktion "PostQuitMessage" aufruft. Als Ergebnis liefert die Funktion "GetMessage" den Wert Null zurück.

Was würde passieren, wenn wir den Aufruf von "PostQuitMessage(0)" entfernen würden? Zwar würde das Fenster verschwinden, aber wenn Sie das Programm aus Delphi heraus starten, dann würde der Debugger immer noch laufen. Mit anderen Worten: das Programm läuft weiter und kommt aus der Nachrichtenschleife nicht mehr heraus. Beenden lässt es sich dann nur über den Taskmanager, bzw. (in der Delphi-IDE) durch ein erneutes Kompilieren.

1.1.6. Die Nachrichtenschleife

In der Nachrichtenschleife werden alle Nachrichten gesammelt. Wird das Fenster geschlossen, liefert "GetMessage" den Wert "WM_DESTROY". Dies Nachricht wird an die Fensterfunktion "WndProc" weitergeleitet und dort mit "PostQuitMessage(0);" beantwortet. Dadurch wird in "GetMessage" der Rückgabewert Null (**false**) erzeugt, der die **while**-Schleife beendet und das Programm dann tatsächlich beendet.:

```
while GetMessage(msg,0,0,0) do
  begin
    TranslateMessage(msg);
    DispatchMessage(msg);
  end;
```

An dieser Stelle soll Ihnen ein Auszug aus der Hilfe den Umgang mit "GetMessage" verdeutlichen:

```
BOOL GetMessage(
  LPMSG lpMsg,          // address of structure with message
  HWND hWnd,            // handle of window
  UINT wMsgFilterMin,   // first message
  UINT wMsgFilterMax    // last message
);
```

Ist `hWnd` auf Null gesetzt, erhält die Funktion die Nachrichten aller Fenster, die zum aufrufenden Thread gehören. Mit Hilfe von `wMsgFilterMin` und `wMsgFilterMax` kann man die eingehenden Nachrichten filtern.

Zu Testzwecken kommentieren wir die beiden Zeilen im Schleifenrumpf des Beispielprogramms einfach mal aus und beobachten das Ergebnis. Wir stellen fest, dass unser Programm gar nicht mehr reagiert. Wie sollte es auch? "`GetMessage`" fängt zwar die Nachrichten ab, kann sie aber (mangels Funktionen) nicht weiterreichen. Es findet keine Verarbeitung der Nachrichten statt, und das Programm hängt in der Endlosschleife fest.
Wir brauchen also zumindest die Funktion "`DispatchMessage`", denn diese gibt die Nachrichten geordnet an die Fensterfunktion weiter. "`TranslateMessage`" übersetzt virtuelle Tastaturcodes - da wir in unserem Beispiel aber keine Tastatureingaben verarbeiten, könnten wir ebenso gut auf diese Funktion verzichten.

Die `Msg`-Struktur ist die Schnittstelle zu den Windows-Nachrichten. Innerhalb dieser Struktur befinden sich alle notwendigen Informationen, die die Nachricht beschreiben. Sie sehen dies sehr schön in der Funktion "`WndProc`", wo auf die verschiedenen Nachrichten reagiert wird. Schauen Sie sich doch bitte einmal diesen Auszug aus der besagten Funktion an:

```
WM_LBUTTONDOWN:
  begin
       ahdc := GetDC(hWnd);
       xPos := LoWord(lParam);
       ypos := HiWord(lParam);
       wvsprintf(buffer, 'Fensterhandle: %d',  PChar(@hWnd));
       wvsprintf(buffer1, ', Message: %d', PChar(@uMsg));
       lstrcat(buffer, buffer1);
       wvsprintf(buffer1, ', wParam: %d', PChar(@wParam));
       lstrcat(buffer, buffer1);
       wvsprintf(buffer1, ', LoWord(lParam) x-Pos: %d', PChar(@xpos));
       lstrcat(buffer, buffer1);
       wvsprintf(buffer1, ', HiWord(lParam) y-Pos: %d', PChar(@ypos));
       lstrcat(buffer, buffer1);
       TextOut(ahdc, 20, 20, buffer, Length(buffer));
       ReleaseDC(hWnd, ahdc);
  end
```

Hier wird auf die Nachricht reagiert, die beim Klick der linken Maustaste im Anwendungsbereich des Fensters entsteht. Als Ergebnis sehen Sie ein paar numerische Werte, von denen offensichtlich nur "`Message`" immer den gleichen Wert hat. 513 in diesem Fall, was anscheinend mit der Nachricht "`WM_LBUTTONDOWN`" identisch ist. Testen Sie es und ersetzen Sie im o.g. Quellcode den Bezeichner der Nachricht durch den numerischen Wert "513". Das Programm funktioniert danach weiterhin, denn tatsächlich entspricht dieser numerische Wert der Nachricht "`WM_LBUTTONDOWN`".

Windows scheint also nur aus Zahlen zu bestehen, die allerdings - Gott sei Dank! - zum einfacheren Programmieren in der Unit "`windows.pas`" als Konstanten definiert sind.
Sie sollten allerdings der Versuchung widerstehen, die numerischen Werte zu verwenden. Es ist nämlich nie ausgeschlossen, dass in einer künftigen Version von Windows ganz andere Werte benutzt werden. Sie können zwar davon ausgehen, dass (um bei Delphi zu bleiben) Borland in diesem Fall eine angepasste Version der betroffenen Units veröffentlichen würde, aber das bezieht sich ja nur auf die Werte für die Konstanten. Wenn in Ihrem Programm aber der numerische Wert steht, können Ihre Units so aktuell wie nur möglich sein - das Programm wird trotzdem nicht mehr wie gewohnt funktionieren.

Um auf das Beispiel mit der linken Maustaste zurückzukommen -
Starten Sie bitte das Beispielprogramm und drücken Sie die linke Maustaste mehrmals. Halten Sie dazu bitte auch die rechte Maustaste oder STRG oder Shift gedrückt und beobachten Sie den Wert von `wParam`. Sie werden feststellen, dass sich dieser ändert - je nachdem, welche zusätzliche Taste Sie noch gedrückt halten.

Das erlaubt Ihnen die ganz gezielte Abarbeitung unter verschiedenen Bedingungen. Der Wert von `wParam` muss also nicht immer Eins sein. Wenn Sie z.B. Shift und die linke Maustaste drücken, würde das Ergebnis Fünf sein. Die Hilfe gibt Ihnen genauere Informationen, für uns soll noch eine Erweiterung interessant sein. Ändern wir den o.g. Code also einmal so ab, dass er die Informationen nur noch liefert, wenn der Anwender sowohl die linke Maustaste als auch Shift drückt:

```
WM_LBUTTONDOWN:
  if(MK_LBUTTON or MK_SHIFT = wParam) then
    begin
      // usw.
    end;
```

1.1.7. Zwei Fenster erzeugen

Mit dem Wissen, das wir nun haben, stellt es uns vor kein Problem, zwei Fenster anzuzeigen. Wir brauchen dazu einfach nur zwei Nachrichtenfunktionen - für jedes Fenster eine. Alles andere können wir 1:1 aus den vorangegangenen Kapiteln übernehmen.

Also erzeugen wir erst einmal unsere zwei Fenster:

```
{Struktur mit Infos für Fenster 1 füllen}
wc.hInstance  := hInstance;
wc.hIcon      := LoadIcon(hInstance,MAKEINTRESOURCE(100));
wc.hCursor    := LoadCursor(0, IDC_ARROW);

{Fenster 1 registrieren}
RegisterClassEx(wc);

{Fenste 1 erzeugen und hWnd1 zuweisen}
CreateWindowEx(0, ClassName1, Window1Name, WS_VISIBLE or
  WS_OVERLAPPEDWINDOW,
  CW_USEDEFAULT, CW_USEDEFAULT, WindowWidth1, WindowHeight1, 0, 0, hInstance,
  nil);

{Struktur mit Infos für Fenster 2 füllen}
wc.hInstance  := hInstance;
wc.lpfnWndProc := @Wnd2Proc;  //Fensterfunktion für Fenster 2
wc.hIcon      := LoadIcon(0, IDI_INFORMATION);
wc.hCursor    := LoadCursor(0, IDC_ARROW);
wc.lpszClassName := ClassName2;   //Klassenname für Fenster 2

{Fenster 2 registrieren}
RegisterClassEx(wc);
```

Was fehlt in diesem Code?
Genau. Das zweite Fenster wurde zwar registriert, aber es wurde noch nicht mit "`CreateWindowEx`" erzeugt. Das machen wir erst auf Knopfdruck, nachdem der Anwender den Button angeklickt hat.

```
WM_COMMAND:
  begin
    if hiword(wParam) = BN_CLICKED then
      case loword(wParam) of
        IDC_BUTTON1:
          {Fenster 2 erzeugen und anzeigen}
          hwnd2 := CreateWindowEx(0, ClassName2, Window2Name,
            WS_OVERLAPPEDWINDOW or WS_VISIBLE, 40, 10,
            300, 200, hWnd1, 0, hInstance, nil);
      end;
  end;
```

Das komplette Beispiel für zwei Fenster finden Sie bei den Demos.

1.2. Textausgabe in Labels

1.2.1. Das Label erzeugen

In diesem Tutorial konzentrieren wir uns auf Labels und das Ändern der Schriftart in solchen. Auf dem herkömmlichen Weg mit der VCL ist es ja recht einfach, solche statischen Texte auf das Formular zu setzen und zu beschriften. Ohne die VCL müssen wir uns dazu u.a. etwas näher mit der Funktion "`SendMessage`" befassen. Unsere Labels erstellen wir am besten wenn unser Fenster die Nachricht "`WM_CREATE`" erhält. Dazu die (freie) Übersetzung aus dem MSDN:

MSDN:

WM_CREATE wird gesendet, wenn die Anwendung die Erzeugung eines Fensters durch "`CreateWindowEx`" oder "`CreateWindow`" anfordert. Die Fensterfunktion des neuen Fensters erhält diese Nachricht, nachdem das Fenster erzeugt wurde, jedoch bevor es angezeigt wird.

Einen besseren Ort kann es also kaum geben.

Nun würde Windows nicht "Windows" heißen, wenn sein Hauptbestandteil nicht Fenster wären. Und so merkwürdig sich das manchmal auch anhören mag - wir sollten uns darüber im Klaren sein, dass es sich bei fast jedem Element um ein Fenster handelt. Auch unser Label ist so ein Fenster. Aus dem Grund schauen wir uns an dieser Stelle die Definition von "`CreateWindowEx`" an:

```
HWND CreateWindowEx(
  DWORD dwExStyle,      // extended window style
  LPCTSTR lpClassName,  // pointer to registered name
  LPCTSTR lpWindowName, // pointer to window name
  DWORD dwStyle,        // window style
  int x,                // horizontal position of window
  int y,                // vertical position of window
  int nWidth,           // window width
  int nHeight,          // window height
  HWND hWndParent,      // handle to parent or owner window
  HMENU hMenu,          // handle to menu, or child-window identifier
  HINSTANCE hInstance,  // handle to application instance
  LPVOID lpParam        // pointer to window-creation data
);
```

Wir benötigen also:

- einen Zeiger auf eine registrierte Fensterklasse
- einen Zeiger auf einen Fensternamen
- das Handle auf das übergeordnete Fenster
- einen Bezeichner für das untergeordnete Fenster (Child window)
- ein Handle auf die Anwendungsinstanz

Aber erzeugen wir erst einmal unser Fenster mit den Labels, und schauen wir uns dann an wo was im Code zu finden ist. Wie auch bei unserem Hauptfenster steht der erste Parameter für den erweiterten Fenster-Stil. Es bietet sich hier wieder an an mit verschiedenen Werten zu experimentieren und die Ergebnisse zu beobachten.

```
hwndLabel1 := CreateWindowEx(0,
```

Der zweite Parameter ist ein Zeiger auf die registrierte Fensterklasse, die letztlich für das verantwortlich ist, was wir mit dem Aufruf erzeugen (wollen) - ob nun Schaltfläche, Editfeld oder eben Label ... Näheres findet man im MSDN unter dem Stichwort "`CreateWindow`".

```
  'STATIC',
```

Der dritte Parameter ist augenscheinlich unser Text, den wir anzeigen wollen.

```
  'Label mit Systemschrift',
```

Der vierte Parameter bestimmt in diesem Fall das Aussehen unseres Labels, wobei zu bemerken ist, dass jede registrierte Fensterklasse noch ihre zusätzlichen Stile hat. Der Text unseres Labels kann etwa mit der Eigenschaft `SS_CENTER` zentriert werden. Nähere Informationen gibt es dazu natürlich auch im MSDN.
Wichtig ist aber die Eigenschaft `WS_CHILD`, mit der Windows weiß, dass unser Label ein untergeordnetes Fenster ist. Und um es auch gleich sichtbar zu machen, sollten wir `WS_VISIBLE` nicht vergessen.

```
  WS_VISIBLE or WS_CHILD,
```

Die nächsten vier Parameter bestimmen natürlich wieder Ursprungskoordinaten sowie Länge und Breite unseres Labels. Diese Angaben beziehen sich auf die linke obere Ecke unseres Hauptfensters.

```
  15, 25, 160, 20,
```

Das Handle dieses Fensters geben wir auch gleich als nächsten Parameter an. Die Nachrichtenfunktion "`WndProc`" enthält in ihren Parametern das gültige Handle unseres Fensters, so dass wir es hier verwenden können.

```
  hWnd,
```

Der zehnte Parameter definiert normalerweise das Menü unseres Fensters. Da wir es aber hier mit einem untergeordneten (Child-)Fenster zu tun haben, geben wir stattdessen einen eindeutigen Bezeichner an, der unser Label identifiziert. Auf diese Weise können Nachrichten später eindeutig zugeordnet werden, und wir haben die Möglichkeit, z.B. den Text unseres Labels zu ändern. Sie sollten für solche Zwecke Konstanten definieren (wie im Beispiel "`IDC_LABEL1`"), die Sie im ganzen Programm verwenden können. Eine mögliche Änderung des Wertes bringt dann nicht Ihr ganzes Programm durcheinander.

```
  IDC_LABEl1,
```

Der elfte Parameter erhält das Handle auf unsere Anwendungsinstanz. In der Regel ist das der Wert der Fensterklassen-Eigenschaft "`wc.hInstance`", kurz: hier also `hInstance`.

```
  hInstance,
```

Den letzten Parameter können wir hier ignorieren und setzen ihn daher auf `nil`.

```
  nil);
```

1.2.2. Schriftart erzeugen und zuweisen

Unser Label wird normalerweise mit der Standardschriftart des Systems beschriftet, was nicht immer gewünscht ist und gut aussehen muss. Um eine eigene Schriftart zu benutzen, bedienen wir uns der Funktion "`CreateFont`":

```
MyFont := CreateFont(FontSize, 0, 0, 0, 0, 0, 0, 0, ANSI_CHARSET,
  OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
  DEFAULT_PITCH, FontName);
```

Eine genaue Beschreibung finden Sie in der Hilfe oder im MSDN. Soviel sei gesagt: als Rückgabewert liefert die Funktion ein Handle auf eine logische Schrift. Der erste Parameter gibt die Schrifthöhe an und sollte immer einen negativen Wert haben, und der letzte Parameter definiert den Namen unserer Schrift (eine der vielen installierten Schriftarten kann hier benutzt werden). Der Einfachheit halber habe ich beides auch wieder als Konstanten deklariert.

Nun ist aber nicht nur Windows in der Lage Nachrichten zu senden. Auch wir können Nachrichten verschicken. In diesem Fall müssen wir das sogar tun, wenn wir unserem Label die neu definierte Schriftart zuweisen wollen. Wir benutzen dazu die Nachricht "`WM_SETFONT`". Unser Ziel erreichen wir dabei mit einer sehr wichtigen Funktion, die wir in vielen nonVCL-Projekten noch brauchen werden - "`SendMessage`". Diese Funktion hat vier Parameter. Da wir eine Nachricht an ein Fenster schicken wollen, benötigen wir zuerst natürlich dessen Handle. An zweiter Stelle kommt dann die gewünschte Nachricht. Lediglich die beiden letzten Parameter bedürfen einer näheren Erklärung: Jede Nachricht hat zwei zusätzliche Parameter, die sie weitergeben kann. Manchmal ein Integerwert zur Statusänderung, manchmal ein als Integer konvertierter Zeiger auf Text usw. (An geeigneter Stelle kommen wir noch dazu.) In einigen Fällen werden einer oder

auch beide Parameter nicht genutzt und stattdessen mit Null angegeben.
Die beste Anlaufstelle sollte für Sie hier wieder die Hilfe oder das MSDN sein, denn hier finden Sie die Nachrichten inkl. der Parameter, die benötigt werden.

Um bei unserem Font-Beispiel zu bleiben - unser Aufruf müsste so aussehen:

```
if MyFont <> 0 then
  SendMessage(hwndLabel2, WM_SETFONT, Integer(MyFont), Integer(true));
```

SendMessage-Definition

```
LRESULT SendMessage(
  HWND hWnd,                            // handle of destination window
  UINT Msg,                             // message to send
  WPARAM wParam,                        // first message parameter
  LPARAM lParam                         // second message parameter
);
```

WM_SETFONT-Definition

```
WM_SETFONT
  wParam = (WPARAM) hfont;              // handle of font
  lParam = MAKELPARAM(fRedraw, 0);      // redraw flag
```

1.2.3. Bitmaps im Label anzeigen

Das Label lässt sich auch als Grundlage zur Anzeige von Bitmaps verwenden - wie man es von der **TImage**-Komponente der VCL kennt. Als Beispiel erzeugen wir ein zweites Control, auf der Grundlage des Labels, mit zwei erweiterten Stilelementen:

```
hwndImage := CreateWindowEx(0, 'STATIC', '',
  WS_VISIBLE or WS_CHILD or {neue Stilattribute ->} SS_BITMAP or
SS_REALSIZEIMAGE,
  9, 6, {w & h werden ignoriert ->} 0, 0, hWnd, IDC_IMAGE, hInstance, nil);
```

Die Attribute `SS_BITMAP` und `SS_REALSIZEIMAGE` geben an, dass eine Bitmap aus den Ressourcen im Control angezeigt werden soll, und dass die Größe der Bitmap nicht geändert wird, nachdem sie geladen oder gezeichnet wurde. Ist die Grafik größer als der zur Verfügung stehende Bereich, wird sie entsprechend abgeschnitten.

Im nächsten Schritt holen wir die Bitmap aus den Ressourcen und zeigen Sie im Control an:

```
hBmp := LoadBitmap(hInstance,MAKEINTRESOURCE(300));
SendMessage(hwndImage,STM_SETIMAGE,IMAGE_BITMAP,hBmp);
```

Dazu nutzen wir die Nachricht "`STM_SETIMAGE`", mit der wir auch Symbole und Cursor anzeigen lassen können.

STM_SETIMAGE-Definition

```
STM_SETIMAGE
  wParam = (WPARAM) fImageType;         // image-type flag =
                                        //   IMAGE_BITMAP
                                        //   IMAGE_CURSOR
                                        //   IMAGE_ENHMETAFILE
                                        //   IMAGE_ICON
  lParam = (LPARAM) (HANDLE) hImage;    // handle to the image
```

Nachteilig ist aber, dass (in diesem Fall) die Bitmap Teil des Programms sein muss. Je nach Größe der Grafik nimmt also auch der Umfang des kompilierten Programms zu. Die Funktion "`LoadImage`" ist daher vorzuziehen, denn hier können wir einen Dateinamen angeben. Die Bitmap muss nicht mit kompiliert werden, und das Programm bleibt klein:

```
hBmp := LoadImage(hInstance,         // Anwendungsinstanz
                  'image.bmp',       // Dateiname
                  IMAGE_BITMAP,      // Typ
                  165,314,           // Breite, & Höhe
                  LR_LOADFROMFILE); // notwendiges Flag
```

In der Hilfe und im MSDN finden Sie die Parameter genauer erklärt. Hier soll nur wichtig sein, dass Sie das Flag `LR_LOADFROMFILE` setzen müssen, und dass Sie den Dateinamen dann als zweiten Parameter angeben. Sie sollten zur Sicherheit aber den kompletten Pfadnamen angeben.

Und vergessen Sie bitte nicht, die Bitmap beim Beenden des Programms wieder freizugeben:

```
DeleteObject(hBmp);
```

1.3. Schaltflächen

1.3.1. Buttons erzeugen

Kommen wir nun zum Volkssport: Buttonklicken ... :o)

Auch hier müssen wir zunächst ein Fenster erzeugen. Dabei richten wir uns einfach nach dem Beispiel des Labels, benutzen aber die registrierte Fensterklasse `BUTTON`, und schon haben wir unseren ersten Button erstellt:

```
hwndButton := CreateWindowEx(WS_EX_CLIENTEDGE, 'BUTTON', 'Beenden',
  WS_VISIBLE or WS_CHILD, 45, 40, 100, 25, hWnd, IDC_BUTTON, hInstance,
  nil);
```

Die zusätzlichen Fensterstile für Buttons finden Sie - wie gehabt - in der Hilfe und im MSDN unter dem Stichwort "`CreateWindow`".

1.3.2. Auf Button-Klicks reagieren

Ach wie schön kann doch die VCL sein - hier packen Sie Ihren Button auf Ihr Formular, klicken doppelt drauf und schreiben den Code in den Quelltexteditor, der beim Klick ausgeführt werden soll. Ohne die VCL ist es zwar nicht ganz so einfach, aber wenn man das Prinzip einmal verstanden hat, dann verzieht man das Gesicht auch nicht mehr. :o)

In diesem Fall benötigen wir die Nachricht "`WM_COMMAND`", die gesendet wird, wenn der Anwender einen Menübefehl aufruft, einen Shortcut wählt ... oder wenn ein Element eine Benachrichtigung ("notification message") an sein übergeordnetes Fenster schickt:

```
WM_COMMAND
  wNotifyCode = HIWORD(wParam); // notification code
  wID = LOWORD(wParam);         // item, control, or accelerator identifier
  hwndCtl = (HWND) lParam;      // handle of control
```

wNotifyCode

Wert des höherwertigen Wortes von `wParam`. Enthält den Benachrichtigungscode wenn die Nachricht von einem Control kommt. Ist die Nachricht von einem Shortcut, dann ist der Parameter 1. Ist die Nachricht von einem Menü, ist der Parameter 0.

wId

Wert des niederwertiges Wortes von `wParam`. Spezifiziert die ID des Menüitems, Controls oder Shortcuts.

hWndCtl

Wert von `lParam`. Handle zum Control, das die Nachricht gesendet hat (wenn die Nachricht von einem Control ist). Andernfalls ist dieser Parameter Null.

Wenn also die Nachricht "`WM_COMMAND`" auftritt, dann wird zuerst geprüft, welches Ereignis sie ausgelöst hat. Handelt es sich um ein Button-Klickereignis ("`BN_???`"-Messages), dann wäre also geklärt, dass ein Button gedrückt wurde. Bleibt die Frage: Welcher war es? Dazu wird das niederwertige Wort von `wParam` geprüft, das die ID enthält.
Da der Benutzer in unserem Beispiel damit rechnet, dass dieser Button das Programm beendet, wollen wir ihm den Gefallen mal tun. Wir senden also die Nachricht "`WM_DESTROY`" (alternativ geht auch "`WM_CLOSE`"), und das Programm beendet sich. Im Code sieht das dann so aus:

```
WM_COMMAND:
  begin
    if hiword(wParam) = BN_CLICKED then
      case loword(wParam) of
        IDC_BUTTON: SendMessage(hwnd, WM_DESTROY, 0, 0);
      end;
  end;
```

1.3.3. Bitmaps und Icons auf Buttons anzeigen

Als letzten Schritt wollen wir einen zweiten Button mit einer Bitmap versehen. Ein solcher Button wird auf die selbe Art wie ein normaler Button erzeugt. Neu ist nur das Stilattribut `BS_BITMAP` (für Bitmaps) bzw. `BS_ICON` (für Symbole):

```
hwndBmpButton := CreateWindowEx(0, 'BUTTON', 'Button', WS_VISIBLE or
  WS_CHILD or {neuer Stil ->} BS_BITMAP, 45, 60, 100, 25, hWnd, IDC_BMPBUTTON,
hInstance,
  nil);
```

Um die Bitmap aus den Programmressourcen zu laden, benutzen wir "`LoadBitmap`".

```
hwndBMP := LoadBitmap(hInstance, MAKEINTRESOURCE(101));
```

Rückgabewert dieser Funktion ist ein Handle auf die Bitmap:

LoadBitmap-Definition

```
HBITMAP LoadBitmap(
  HINSTANCE hInstance,    // Anwendungsinstanz
  LPCTSTR lpBitmapName    // Name der Bitmapressource
);
```

Ein Icon wird auf ähnliche Weise geladen, wobei man hier auch noch den Vorteil hat, transparente Grafiken nutzen zu können. Das folgende Beispiel zeigt, wie man ein 16x16 großes Icon lädt:

```
hwndICO := LoadImage(hInstance,MAKEINTRESOURCE(101),IMAGE_ICON,
  16,16,LR_DEFAULTCOLOR);
```

LoadImage-Definition

```
HANDLE LoadImage(
    HINSTANCE hinst,    // Anwendungsinstanz
    LPCTSTR lpszName,   // Ressourcen-ID der Bitmap, oder Dateiname
    UINT uType,         // Imagetyp
    int cxDesired,      // Breite
    int cyDesired,      // Höhe
    UINT fuLoad         // Flags
);
```

Bitmap oder Icon werden dann mit der Nachricht "`BM_SETIMAGE`" auf dem Button platziert:

```
SendMessage(hwndBmpButton, BM_SETIMAGE, IMAGE_BITMAP, hwndBMP);
```

Im `wParam`-Parameter geben Sie an, ob das Bild eine Bitmap (`IMAGE_BITMAP`) oder ein Symbol (`IMAGE_ICON`) ist. Das Handle der Bitmap, bzw. des Icons wird im letzten Parameter angegeben.

BM_SETIMAGE-Definition

```
BM_SETIMAGE
  wParam = (WPARAM) fImageType;       // image-type flag
  lParam = (LPARAM) (HANDLE) hImage;  // handle to the image
```

Im Beispielprogramm kann mit Hilfe eines Compilerschalters entschieden werden, ob der Button eine Bitmap oder ein Icon verwenden soll.

1.4. Checkboxen und Radiobuttons

1.4.1. Checkboxen und Radiobuttons erzeugen

Auch Checkboxen und Radiobuttons sind im Grunde nur Schaltflächen, nur dass sie zusätzliche Stileigenschaften aufweisen, die sie von gewöhnlichen Buttons unterscheiden.

Eine Checkbox erstellt man mit dem Stil `BS_CHECKBOX` oder `BS_AUTOCHECKBOX`. Der Unterschied besteht darin, dass sich der Programmierer beim Stil `BS_CHECKBOX` selbst darum kümmern muss, ob die Checkbox markiert oder nicht markiert dargestellt wird. Das Beispielprogramm benutzt diesen Stil, um zu zeigen wie man den Status abfragt und ändert:

```
hwndChkBox := CreateWindowEx(0, 'BUTTON', 'Checkbox', WS_VISIBLE or
  WS_CHILD or {neuer Stil ->} BS_CHECKBOX, 10, 20, 90, 25, hWnd, IDC_CHKBOX,
  hInstance, nil);
```

Der Code für einen Radiobutton sieht ähnlich aus; natürlich werden ein anderer Klassenname und andere Stilattribute verwendet. Hier habe ich übrigens den Stil `BM_AUTORADIOBUTTON` gewählt. Damit kümmert sich dann das System um die Anzeige des Status. Allerdings müssen sich die benutzten Radiobuttons dann in der gleichen Gruppe befinden:

```
hwndOpt1 := CreateWindowEx(0, 'BUTTON', 'Radiobutton1', WS_VISIBLE or
  WS_CHILD or {neuer Stil ->} BS_AUTORADIOBUTTON, 25, 75, 125, 25, hWnd,
IDC_OPT1,
  hInstance, nil);
```

1.4.2. Das Klickereignis von Checkboxen und Radiobuttons

Das Markieren oder Entfernen der Markierung entspricht wieder einem ganz normalen Button-Klickereignis und wird daher auch über "`WM_COMMAND`" bearbeitet. Der Status wird mit der Button-Nachricht "`BM_GETCHECK`" abgefragt. Rückgabewert ist dann entweder `BST_CHECKED` (Haken gesetzt) oder `BST_UNCHECKED` (Haken entfernt). Im Beispiel wird das Ergebnis einer `bool`-Variablen zugeordnet

```
bCBFlag := (SendMessage(hwndChkBox,BM_GETCHECK,0,0) = BST_CHECKED);
```

und in negierter Form an die Checkbox zurückgegeben, um den Status mit der Nachricht "`BM_SETCHECK`" zu ändern:

```
SendMessage(hwndChkBox,BM_SETCHECK,CheckFlags[not(bCBFlag)],0);
```

Bei Radiobuttons ist ebenso vorzugehen, denn sie senden und empfangen die selben Nachrichten wie Checkboxen.

1.5. Arbeiten mit Eingabefeldern

1.5.1. Eingabefelder erzeugen

Der Fokus dieses Tutorials liegt auf den wichtigsten Funktionen von Eingabefeldern. Sie sind zu komplex, um jedes Detail zu behandeln. Am häufigsten wird man wohl etwas in sie schreiben bzw. etwas aus ihnen auslesen wollen. Und genau das wird hier demonstriert.

Auch ein Eingabefeld wird mit der Funktion "CreateWindowEx" erstellt. Und auch hier gibt es spezielle Stilattribute, die Sie nutzen können, und die Sie wie immer in der Hilfe oder im MSDN finden. Unser Beispielprogramm erzeugt ein Editfeld z.B. mit folgendem Aufruf:

```
hwndEdit1 := CreateWindowEx(WS_EX_CLIENTEDGE, 'EDIT', 'Edit1', WS_VISIBLE or
  WS_CHILD or ES_NOHIDESEL, 10, 20, 400, 20, hWnd, IDC_EDIT1, hInstance, nil);
```

Als Besonderheit ist die Eigenschaft ES_NOHIDESEL zu nennen. Normalerweise versteckt das System den aktuell markierten Text, wenn Sie den Fokus wechseln und vom Eingabefeld zu einem anderen Control gehen. Erst wenn das Editfeld den Fokus zurückerhält, sehen Sie die Markierung auch wieder. Mit der o.g. Eigenschaft können Sie dieses Verhalten ausschalten, so dass der markierte Text auch dann markiert bleibt, wenn Sie ein anderes Control benutzen. Weitere Eigenschaften finden Sie wie ebenfalls in der Hilfe und im MSDN.

1.5.2. Text in Eingabefeldern kopieren

Mit der Schaltfläche "Text kopieren" in unserem Beispielprogramm wird der Text des oberen Eingabefeldes in das untere kopiert. Dazu wird der vorhandene Text mit der Nachricht "WM_GETTEXT" in einen Puffer gelesen:

```
SendMessage(hwndEdit1, WM_GETTEXT, 1024, Integer(@buffer));
```

und dann einfach nur mit dem Gegenstück "WM_SETTEXT" in das untere Feld geschrieben:

```
SendMessage(hwndEdit2, WM_SETTEXT, 0, Integer(@buffer));
```

WM_GETTEXT-Definition

```
WM_GETTEXT
  wParam = (WPARAM) cchTextMax;   // number of characters to copy
  lParam = (LPARAM) lpszText;     // address of buffer for text
```

WM_SETTEXT-Definition

```
WM_SETTEXT
  wParam = 0;                     // not used; must be zero
  lParam = (LPARAM)(LPCTSTR)lpsz; // address of window-text string
```

Als Alternative zu diesen beiden Nachrichten ließen sich auch noch "GetWindowText" und "SetWindowText" verwenden:

```
GetWindowText(hwndEdit1,@buffer,1024);
SetWindowText(hwndEdit2,@buffer);
```

GetWindowText-Definition

```
int GetWindowText( HWND hWnd, // handle to window or control LPTSTR lpString,
// address of string int nMaxCount // address of string );
```

SetWindowText-Definition

```
BOOL SetWindowText(
  HWND hWnd,                    // handle to window or control
  LPCTSTR lpString              // address of string
);
```

Ich möchte an dieser Stelle noch einmal erwähnen, dass auch ein Eingabefeld ebenso ein Fenster wie das Hauptfenster ist. Es wurde nur mit anderen Stilen erzeugt. Warum ich das noch sage? Setzen sie doch einfach mal anstelle des Editfeld-Handles das Handle des Labels ein und beobachten Sie was dann passiert ... Noch interessanter wird es, wenn sie die Prozedur um den Parameter "`hWnd`" erweitern und diesen dann einsetzen. Was geschieht? Wir können in das obere Eingabefeld reinschreiben, was wir wollen - im unteren Eingabefeld erscheint entweder der Text aus dem Label oder unser Titelzeilentext.

Mit der Editfeld-Nachricht "`EM_LINELENGTH`" wird einfach nur ermittelt wie viele Zeichen in der angegebenen Zeile sind. Im ersten Parameter wird die Zeile angegeben, von der aus ermittelt werden soll. Bei einzeiligen Eingabefeldern wie im Beispiel ist dieser Wert Null. Rückgabewert ist dann die Anzahl der vorhandenen Zeichen:

```
Textlen := SendMessage(hwndEdit1, EM_LINELENGTH, 0, 0);
```

EM_LINELENGTH-Definition

```
EM_LINELENGTH
  wParam = (WPARAM) ich;        // character index
  lParam = 0;                   // not used; must be zero
```

Nebenbei gesagt: mit dem Stil "`ES_MULTILINE`" können Sie ein mehrzeiliges Eingabefeld erzeugen. Für einen einfachen Texteditor beispielsweise. Hier wäre es dann möglich, mit Hilfe des `ich`-Parameters (im `wParam`) die Zeichenanzahl ab einer bestimmten Zeile zu bestimmen.

1.5.3. Text im Eingabefeld markieren

Um Text zu markieren, tragen Sie in die beiden kleinen Editfelder des Beispielprogramms bitte die Start- und Endposition ein. Benutzen Sie dann den Button "Markieren", und der Text im oberen Eingabefeld wird - gemäß Ihren Angaben - markiert. Wenn nichts passiert, dann haben Sie wahrscheinlich keine Angaben gemacht. Wie funktioniert es?

Mit der schon bekannten Nachricht "`WM_GETTEXT`" werden zunächst Ihre Angaben aus den beiden kleinen Eingabefeldern ausgelesen und mit Hilfe von "`val`" in numerische Werte konvertiert:

```
SendMessage(hwndEdit3, WM_GETTEXT, 1024, Integer(@buffer));
val(buffer, SelStart, code);
```

Wenn auf diese Weise Start- und Endpunkt klar sind, müssen beide Werte nur mit der Eingabefeld-Nachricht "`EM_SETSEL`" an das Eingabefeld übermitteln, wodurch die Markierung dann vorgenommen und sichtbar wird:

```
SendMessage(hwndEdit1, EM_SETSEL, SelStart, SelEnd);
```

EM_SETSEL-Definition

```
EM_SETSEL
  wParam = (WPARAM) (INT) nStart;     // starting position
  lParam = (LPARAM) (INT) nEnd;       // ending position
```

Ausgewählten Text in die Zwischenablage kopieren

Das Kopieren und Einfügen von Text geht am einfachsten mit den Nachrichten "WM_COPY" und "WM_PASTE". Die erste Nachricht kopiert den markierten Text des angegebenen Controls (Fensters, Eingabefeldes ...) mit dem Format "CF_TEXT" in die Zwischenablage:

```
SendMessage(hwndEdit1, WM_COPY, 0, 0);
```

Die zweite Nachricht holt den Text wieder aus der Zwischenablage und trägt ihn in das angegebene Control ein:

```
SendMessage(hwndEdit2, WM_PASTE, 0, 0);
```

Die Werte von wParam und lParam werden hier nicht benötigt und können auf Null gesetzt werden.

1.5.4. Das "OnChange"-Ereignis ohne die VCL

In vielen Programmen sieht man, dass z.B. der Status eines Buttons vom Vorhandensein eines Textes im Eingabefeld abhängig ist. Erst wenn Text vorhanden ist, wird also der Button aktiviert. Im herkömmlichen VCL-Weg eignet sich dazu am besten das "**OnChange**"-Ereignis:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  MachWeiterKnopf.Enabled := (Edit1.Text <> '');
end;
```

Bei der API-Programmierung sind es ein paar Zeilen mehr, aber das Ergebnis ist letzten Endes das selbe. Zuerst erzeugen wir einen deaktivierten Button, wozu wir das Attribut WS_DISABLED benutzen:

```
MachWeiterBtn := CreateWindowEx(0, 'BUTTON', 'Weiter >', WS_VISIBLE
  or WS_CHILD {neuer Stil ->} or WS_DISABLED, 220, 65, 150, 25, hWnd,
IDC_BUTTONADDSTR, hInstance,
  nil);
```

Dann benötigen wir die Nachricht "EN_CHANGE", die als höherwertiger Anteil des wParam-Parameters von "WM_COMMAND" übermittelt wird. Diese Nachricht signalisiert, dass sich im Eingabefeld etwas verändert hat:

```
WM_COMMAND:
  case hiword(wParam) of
    EN_CHANGE:
      if loword(wParam) = IDC_EDIT1 then
        begin
          { ... }
        end;
  end;
```

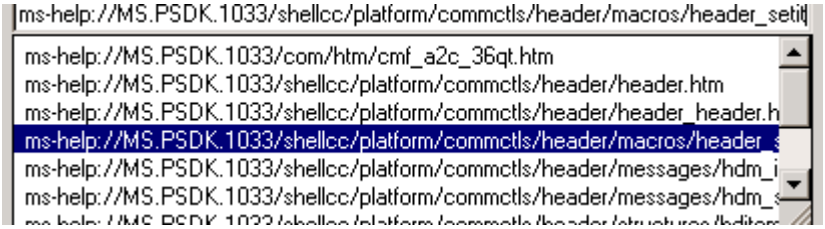
Im niederwertigen Anteil finden Sie die ID des Controls, so dass Sie gezielt prüfen können. Im dritten und letzten Schritt lesen Sie den vorhandenen Text aus dem Control aus und prüfen das Rückgabeergebnis. Dazu können Sie die Nachricht "`WM_GETTEXT`" (in Verbindung mit "`SendMessage`") oder "`GetWindowText`" verwenden. Beide liefern die Anzahl der kopierten Zeichen zurück, so dass Sie davon ausgehen können, dass nur Text vorhanden ist, wenn das Ergebnis größer als Null ist. Abhängig davon aktivieren Sie den Button, oder Sie lassen ihn deaktiviert:

```
EnableWindow(MachWeiterBtn,GetWindowText(hEdit1,buffer,256) > 0);
```

1.5.5. Auto-Vervollständigung aus dem System

In diesem Kapitel soll es nur um die Auto-Vervollständigung gehen, die Ihnen vom System zur Verfügung gestellt wird. Minimal erforderlich ist dazu allerdings der Internet Explorer 5. Das bedeutet, unter Windows 95 und 98 sowie NT4 könnte es möglicherweise zu Problemen kommen, wenn der Anwender keinen oder einen veralteten IE benutzt.

Die Auto-Vervollständigung lässt sich nur für Eingabefelder verwenden. Dazu zählt auch das in eine ComboBoxEx eingebettete Edit-Control. Und sie verhält sich wie die vom System bekannten Funktionen. Das heißt, bei einer ComboBoxEx zeigt Ihnen die Auto-Vervollständigung nicht die Werte an, die in Ihrer ComboBox stehen, sondern Sie sehen die Namen von gestarteten Programmen, die URLs aufgerufener Internetseiten, usw.



In der Praxis lohnt sich diese Shell-Funktion daher eigentlich nur, wenn Sie Ihr Programm mit einer ähnlichen Eingabezeile wie Start/Ausführen ausrüsten wollen und zusätzlich gern Unterstützung vom System hätten.

Die Auto-Vervollständigung besteht aus zwei Teilen: *AutoAppend* bedeutet, dass der von Ihnen eingegebene Text automatisch beim Tippen komplettiert wird (sofern irgendein String in der MRU-Liste der Shell mit den getippten Buchstaben identisch ist). Und als *AutoSuggest* bezeichnet man das kleine Fenster, dass beim Tippen aufklappt und Ihnen die MRU-Liste anzeigt.

Hinweis

> Möglicherweise ist Ihre Delphi-Version zu alt und kennt den Befehl "`SHAutoComplete`" deshalb noch nicht. Für den Fall finden Sie in den Beispielprogrammen die Unit "`ShlObj_Fragment.pas`", die den Befehl samt Flags enthält und auch vom Beispielprogramm ("`Edit_R2.dpr`") benutzt wird.

Um die Auto-Vervollständigung nutzen zu können, muss Ihr Programm beim Start den Befehl "`CoInitialize`", und beim Beenden "`CoUninitialize`" (Unit "`ActiveX`") aufrufen.
Das bedeutet aber auch, dass speziell der letzte Befehl nicht irgendwo vorher ein zweites Mal aufgerufen werden darf. Idealerweise sollten Sie darum so vorgehen:

```
if(CoInitialize(nil) = S_OK) then try
  // Fenster erzeugen, &
  // Nachrichtenbearbeitung starten
finally
  CoUninitialize;
end;
```

und zwischen **try** und **finally** den notwendigen Code zum Erstellen der Anwendung usw. einfügen.

Wenn Sie ein Edit-Control mit der Auto-Vervollständigung ausrüsten wollen, dann benutzen Sie einfach dessen Handle und rufen Sie den schon genannten Befehl "SHAutoComplete" auf:

```
SHAutoComplete(hwndEdit1,SHACF_DEFAULT);
```

Der zweite Parameter kann dabei eins der folgenden Flags sein

Wert	Bedeutung
SHACF_DEFAULT	Identisch mit SHACF_FILESYSTEM + SHACF_URLALL. Kann nicht mit anderen Flags kombiniert werden!
SHACF_FILESYSTEM	Berücksichtigt das Dateisystem (sprich: die MRU-Liste der gestarteten Programme, usw.)
SHACF_URLALL	Berücksichtigt den Verlauf und die zuletzt eingetippten URLs des IE. Entspricht SHACF_URLHISTORY + SHACF_URLMRU
SHACF_URLHISTORY	Berücksichtigt den Verlauf
SHACF_URLMRU	Berücksichtigt die zuletzt eingetippten URLs im IE
SHACF_USETAB	Mit Tab kann nun jeweils ein Item aus der MRU-Liste (*AutoSuggest*) gewählt werden. Ist das Flag nicht gesetzt, gelangt man mit Tab wie gewöhnlich zum nächsten Control. Nur in Kombination mit SHACF_FILESYS* oder SHACF_URL* sinnvoll.

Die folgenden Flags dienen dazu, die IE-Einstellungen in der Registry zu überschreiben. Und sie funktionieren nur in Verbindung mit den SHACF_FILESYS*- oder SHACF_URL*-Flags

SHACF_AUTOAPPEND_FORCE_OFF	unabhängig von der Registryeinstellung wird *AutoAppend* für die Eingabezeile ausgeschaltet
SHACF_AUTOAPPEND_FORCE_ON	unabhängig von der Registryeinstellung wird *AutoAppend* für die Eingabezeile eingeschaltet
SHACF_AUTOSUGGEST_FORCE_OFF	unabhängig von der Registryeinstellung wird *AutoSuggest* für die Eingabezeile ausgeschaltet
SHACF_AUTOSUGGEST_FORCE_ON	unabhängig von der Registryeinstellung wird *AutoSuggest* für die Eingabezeile eingeschaltet

Zu guter Letzt noch ein kleiner Blick darauf, wie man die Auto-Vervollständigung für eine ComboBoxEx benutzt. Die Aufgabe beschränkt sich allerdings nur darauf, das Handle des eingebetteten Edit-Controls zu erhalten. Die ComboBoxEx stellt dazu eine eigene Nachricht zur Verfügung, "CBEM_GETEDITCONTROL", womit auch klar wird, warum die Auto-Vervollständigung nur bei ihr nutzen kann - die normale ComboBox bietet nichts vergleichbares.
Wie dem auch sei, wenn Sie das Handle des Eingabefeldes ermittelt haben, dann entspricht der Aufruf des Befehls dem eingangs gezeigten Beispiel:

```
hEdit := SendMessage(hCBEx,CBEM_GETEDITCONTROL,0,0);
if(hEdit <> 0) then
  SHAutoComplete(hEdit,SHACF_DEFAULT);
```

1.5.6. Spezielles für Windows XP

Unter Windows XP bietet Ihnen das Editfeld ein paar kleine Spezialitäten an. Dazu benötigen Sie allerdings die Common Controls 6.0, die Sie durch die Benutzung eines so genannten Manifests verwenden. Dieses Manifest ist eine XML-Datei mit einem festgelegten Inhalt und sorgt u.a. dafür, dass Ihre Programme den neuen XP-Stil (Themes) verwenden, sofern dieser aktiviert ist.
Außerdem bietet das Manifest aber auch Zugang zu erweiterten Möglichkeiten eines Controls. Die hier beschriebenen Funktionen belegen dies, und auch in späteren Beiträgen wird noch darauf eingegangen werden. Schauen wir uns darum jetzt erst einmal die neuen Funktionen für Editfelder an. Dabei spielt es in dem Fall keine Rolle, ob Sie Themes benutzen oder nicht. Es funktioniert auch in der klassischen Ansicht. Wichtig ist nur die Manifestdatei.

Standardmäßiger Text

Wenn Sie ein Eingabefeld erzeugen, dann haben Sie die Möglichkeit, einen bestimmten Text vorzugeben, beispielsweise

```
hwndEdit1 := CreateWindowEx(WS_EX_CLIENTEDGE,'EDIT','Edit1',WS_VISIBLE or
  WS_CHILD or ES_NOHIDESEL,10,20,400,20,hWnd,IDC_EDIT1,hInstance,nil);
```

Sicher kennen Sie aber auch die Suchfunktion des Windows Explorers. Wenn Sie nach Dateien suchen, dann können Sie im oberen Eingabefeld die gewünschten Dateitypen angeben. Das Feld darunter erlaubt die Suche nach einem Begriff, der in den Dateien enthalten sein muss. Dieses Feld zeigt einen Text an, der standardmäßig grau dargestellt wird. Klicken Sie dann in das Feld, verschwindet der Text. Verliert das Feld den Fokus, erscheint er wieder. Das selbe können Sie mit der Funktion "Edit_SetCueBannerText" auch machen, wobei der erste Parameter das Handle des Eingabefeldes ist, und der zweite der Text, den Sie anzeigen wollen.
Das erweiterte Beispielprogramm "Edit_R2.dpr" demonstriert Ihnen das anhand dieser Zeile:

```
Edit_SetCueBannerText(hwndEdit1,'Geben Sie hier etwas ein');
```

was folgendes Ergebnis hat:

So ein Text macht natürlich nur Sinn, wenn das Eingabefeld leer ist. Er wird auch nur dann angezeigt. Nehmen Sie als Beispiel etwa eine Adressverwaltung. Sie wollen die Daten des Benutzers abfragen und präsentieren ihm deshalb ein kleines Interface mit mehreren Eingabefeldern. Mit Hilfe von "Edit_SetCueBannerText" können Sie auf einfache Weise kleine Beschreibungen anzeigen lassen, die dem Benutzer helfen, den Sinn der Eingabefelder zu erfassen.
Und trotz des angezeigten Textes gilt und ist das Eingabefeld nach wie vor leer.

Anders herum soll es auch gehen: mit der Funktion "Edit_GetCueBannerText" bzw. der Nachricht "EM_GETCUEBANNER" soll man den vorhandenen Text auslesen können. Die Betonung liegt auf "soll", weil sich das PSDK in diesem Fall widerspricht. So steht in der Parameterbeschreibung von "Edit_GetCueBannerText" das folgende:

PSDK (Edit_GetCueBannerText)
Parameters

hwnd
Handle to the edit control.
lpcwText
Pointer to a Unicode string that receives the text that is set as the cue banner.
cchText
Number of WCHARs in the string referenced by lpcwText

Das ist meines Erachtens nach sinnvoll. Der Textpuffer hat eine bestimmte Größe, und auch in anderen Fällen ("GetWindowText", etwa) wird neben dem Puffer auch dessen Größe übergeben. Dementsprechend sieht dann auch die Umsetzung der Funktion aus:

```
#define Edit_GetCueBannerText(hwnd, lpwText, cchText) \
  (BOOL)SNDMSG((hwnd), EM_GETCUEBANNER, (WPARAM)(lpwText), (LPARAM)(cchText))
```

Allerdings heißt es bei der Nachricht "EM_GETCUEBANNER", die hinter der o.g. Funktion steckt:

PSDK (EM_GETCUEBANNER)
Parameters

wParam
Not used; must be zero.
lParam
Pointer to a Unicode string that receives the text set as the textual cue.

Beachten Sie bitte, dass hier gesagt wird, der Wert `wParam` wird nicht verwendet und müsse Null sein. Wenn Sie sich dagegen die o.g. Funktion anschauen, dann werden Sie feststellen, dass der Wert sehr wohl verwendet wird. Offenbar weiß also bei Microsoft die linke Hand nicht was die rechte macht.
Wie dem auch sei, genau besehen besteht eigentlich keine Notwendigkeit, diesen *Cue text* auszulesen. Er dient dem Anwender doch nur als Gedankenstütze, damit er weiß welchen Zweck ein Eingabefeld hat. Und eben diese Aufgabe (dieser Zweck) wird sich in den meisten Fällen wohl auch nicht ändern. Und ich persönlich wüsste auch nicht, wozu man den ausgelesenen *Cue text* benutzen soll. Aber zumindest wollte ich Sie auf den ganz offensichtlichen Widerspruch im PSDK hinweisen.

Balloon-Tipps

Balloon-Tipps sind Tooltipp-ähnliche Popups, die allerdings mehr das Aussehen einer Comics-Sprechblase haben. In den Beiträgen über Tooltipps und die Taskbar Notification Area werden wir noch einmal mit ihnen konfrontiert werden. So bestünde bspw. auch die Möglichkeit, einen Tooltipp zu erzeugen, der in dieser Form dann über dem Editfeld angezeigt wird. Das würde dann auch unter anderen Systemen funktionieren. Und sofern die "`shell32.dll`" aktuell genug ist, ließen sich auch die optischen Spielereien, wie der Cartoonstil bzw. der Titel im Tooltipp verwenden.
Speziell für Windows XP gibt es die beiden Funktionen "`Edit_ShowBalloonTip`" und "`Edit_HideBalloonTip`", mit denen sich diese Tooltipps ohne weitere Umstände anzeigen und wieder verbergen lassen. Aber das funktioniert eben nur unter Windows XP.

Zum Anzeigen eines Tooltipps benötigen wir ein `TEditBalloonTip`-Record, bei dem zuerst die Größe initialisiert wird:

```
ebt.cbStruct := sizeof(ebt);
```

Danach geben wir einen Titel an, der im Tooltipp fett gedruckt dargestellt werden wird

```
ebt.pszTitle := 'Edit-Demo Revision 2.1';
```

Der eigentliche Tooltipp-Text wird dann der Membervariablen `pszText` übergeben:

```
ebt.pszText  := 'Es wurden Zeichen im oberen Edit-Control markiert';
```

Dann haben Sie die Möglichkeit, eins von vier Symbolen auszuwählen. Diese Symbole werden über Konstanten identifiziert, die mit dem Präfix `TTI_` beginnen.

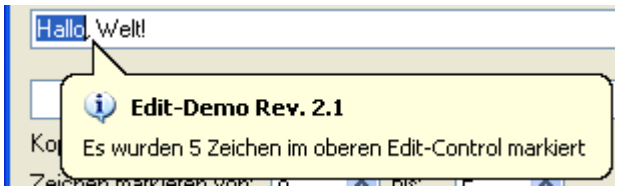
Wert	Bedeutung
TTI_ERROR	Fehlersymbol, rotes Schild mit weißem Kreuz
TTI_INFO	Informationssymbol, Sprechblase mit einem "i" drin
TTI_NONE	kein Symbol
TTI_WARNING	Warnungssymbol, gelbes Dreieck mit Ausrufungszeichen

(Diese Symbole lassen sich auch bei den Tooltipps verwenden.)

```
ebt.ttiIcon  := TTI_INFO;
```

Danach kann das Record mit Hilfe der Funktion "`Edit_ShowBalloonTip`" an das Edit-Control (dessen Handle der erste Parameter ist) übergeben werden

```
Edit_ShowBalloonTip(hwndEdit1,@ebt);
```



Im Gegensatz zu TNA-Programmen verschwinden die Balloontipps der Eingabefelder nicht automatisch. Sie müssen hier gezielt "`Edit_HideBalloonTip`" aufrufen und das Handle des Edit-Controls übergeben:

```
Edit_HideBalloonTip(hwndEdit1);
```

1.6. Arbeiten mit Listboxen

1.6.1. Die Listbox erzeugen

Dieses Tutorial beschäftigt sich mit Listboxen und dem Hinzufügen und Löschen von Einträgen. Auch hier nimmt einem die VCL eine Menge Arbeit ab, aber so schwer ist es auch mit dem API nicht.

An dieser Stelle ganz kurz etwas zur Funktionsweise des Beispielprogramms, da dieses Tutorial auf Teile davon noch näher eingehen wird. Nach dem Start befinden sich schon fünf Einträge in der Liste. Wählt man einen aus, erscheint der Eintrag im Label rechts oben. Es lassen sich neue Einträge über ein Eingabefeld hinzufügen, und es lassen sich markierte Einträge aus der Liste löschen.
Außerdem lassen sich alle Einträge via Button auswählen, und die ausgewählten Einträge können in ein Eingabefeld kopiert werden.

Widmen wir uns also zuerst dem Erzeugen der Listbox. Wie gehabt wird auch sie wie alle anderen Fensterelemente erzeugt. Es gibt ein paar spezielle Stilattribute, von denen einige im folgenden Codeauszug benutzt wurden:

```
hwndListBox := CreateWindowEx(WS_EX_CLIENTEDGE, 'LISTBOX', nil, WS_CHILD
  or WS_VISIBLE or LBS_STANDARD or LBS_EXTENDEDSEL, 10, 10, 200, 230,
  hWnd, IDC_LB, hInstance, nil);
```

Wert	Bedeutung
LBS_EXTENDEDSEL	erlaubt die Auswahl mehrerer Einträge (STRG und Shift werden unterstützt)
LBS_MULTICOLUMN	erstellt eine Listbox mit mehreren Spalten, die horizontal gescrollt werden können
LBS_NOTIFY	das übergeordnete Fenster erhält eine Benachrichtigung wenn auf einen Eintrag geklickt wird
LBS_SORT	die Einträge werden alfabetisch sortiert

Weitere Informationen zu Stilattributen finden Sie im MSDN oder in der Hilfe.

1.6.2. Der Listbox einen Eintrag hinzufügen

Zum Hinzufügen von Einträgen bietet die Listbox die Nachricht "LB_ADDSTRING". Der erste Parameter wird nicht genutzt (ist also Null), und der zweite enthält einen Zeiger auf den Textpuffer.

```
buffer := 'Peter';
SendMessage(hwndListbox, LB_ADDSTRING, 0, Integer(@buffer));
```

Im Beispielprogramm finden Sie eine Funktion, die - via Buttonklick - neue Einträge hinzufügt. Das Prinzip entspricht dabei dem eben gezeigten Muster.

1.6.3. Einen Eintrag aus der Listbox entfernen

Bevor wir einen Eintrag löschen, sollten wir zunächst herausfinden, ob überhaupt Einträge vorhanden, und ob davon min. einer markiert ist. Wir wollen das Programm ja möglichst logisch gestalten und den Anwender nicht verwirren. Wir ermitteln also zunächst mit der Nachricht "LB_GETCOUNT" die Anzahl der Einträge:

```
i := SendMessage(hwndListbox, LB_GETCOUNT, 0, 0);
```

Der Rückgabewert ist entweder die Anzahl der Einträge oder der Wert LB_ERR, der besagt, dass ein Fehler aufgetreten ist. Wenn dieser Fehlercode oder der Wert Null zurückgeliefert werden, sollten wir die Funktion an der Stelle beenden.

Im günstigen Fall können wir uns aber nun den Index des selektierten Eintrages mit der Nachricht "LB_GETCURSEL" holen. Auch hier wird im Fehlerfall LB_ERR zurückgegeben. Ansonsten ist das Ergebnis unser gesuchter Index, und wir können den Eintrag mit "LB_DELETSTRING" löschen:

```
i := SendMessage(hwndListbox, LB_GETCURSEL, 0, 0);
SendMessage(hwndListbox, LB_DELETESTRING, i, 0);
```

1.6.4. Mehrere markierte Einträge auslesen

Die Listbox in unserem Beispielprogramm gestattet die Auswahl mehrerer Einträge. Auch hier wäre eine Fehlerkorrektur empfehlenswert: zuerst sollte man feststellen, wie viele Einträge markiert sind. Dann sucht man den Index des jeweiligen Eintrages und den Text selbst heraus. Das Beispielprogramm enthält eine entsprechende Prozedur, die durch einen Button aufgerufen wird.
Zuerst finden wir mit Hilfe der Nachricht "LB_GETSELCOUNT" die Anzahl der markierten Einträge heraus:

```
CountItems := SendMessage(hwndListbox, LB_GETSELCOUNT, 0, 0);
```

Auf die Weise müssen wir nicht mühsam alle Einträge auf ihren Status prüfen. Wir nutzen stattdessen die Nachricht "LB_GETSELITEMS" und übergeben den eben ermittelten Wert als ersten Parameter. Der zweite ist ein Integer-Array, dass die Indexwerte der markierten Einträge aufnehmen soll. Dieses Array muss groß genug sein. Im Beispielprogramm ist er auf 100 Einträge begrenzt.

```
SendMessage(hwndListbox, LB_GETSELITEMS, CountItems, Integer(@SelItems));
```

Der Rest ist eine **for**-Schleife, in der wir die Indexes durchlaufen, den zugehörigen Text ermitteln und in das Eingabefeld eintragen lassen. Zum Auslesen eines Eintrages nutzen wir die Nachricht "LB_GETTEXT". Diese Nachricht benötigt zwei Parameter. Der erste ist der Index, dessen Text wir ermitteln wollen, und der zweite ist ein Zeiger auf den Puffer, der den Text aufnehmen soll. Das Prinzip ähnelt dabei dem Auslesen eines Editfeldes.

```
for i := 0 to CountItems-1 do
begin
  SendMessage(hwndListbox, LB_GETTEXT, SelItems[i], Integer(@buffer));
  lstrcat(buffer1, buffer);
  lstrcat(buffer1, '; ');
  SetWindowText(hwndEditSelItems, buffer1);
end;
```

Eine elegantere Methode wäre ein dynamisches Array, das sich nach der Anzahl der markierten Einträge richtet und damit auf keine feste Größe (wie 100, in dem Fall) beschränkt ist. Die Umsetzung ist auch recht einfach. Das Array wird wie folgt deklariert:

```
var
  SelItems : array of integer;
```

Wenn wir nun die Anzahl der markierten Einträge ermittelt haben, dann setzen wir die Länge dieses Arrays entsprechend:

```
SetLength(SelItems,CountItems);
```

Der einzige Unterschied zwischen dynamischem und festem Array ist nun der, dass das dynamische Array noch dereferenziert werden muss, damit auch auf das erste Element (mit dem Index Null) zugegriffen wird. Es gibt dafür zwei Möglichkeiten:

1. Sie geben den Index mit an:
   ```
   SendMessage(hwndListbox, LB_GETSELITEMS, CountItems,
   LPARAM(@SelItems[0]));
   ```
2. Sie benutzen diese Anweisung zum Dereferenzieren:
   ```
   SendMessage(hwndListbox, LB_GETSELITEMS, CountItems,
   LPARAM(pointer(@SelItems)^));
   ```

Würden wir das nicht tun, wäre eine Zugriffsverletzung die Folge. Und am Ende müssen wir den belegten Speicher des Arrays wieder freigeben:

```
SetLength(SelItems,0);
```

1.6.5. Die Auswahl programmgesteuert treffen

Soll ein Eintrag programmgesteuert markiert - bzw. seine Markierung aufgehoben - werden, dann benutzt man dazu die Nachricht "LB_SETSEL". Als zweiter Parameter wird dabei der Index des gewünschten Eintrags angegeben, während der erste Parameter den gewünschten Status (**true**, **false**) enthält. Zu beachten ist auch hier wieder, dass der Bool-Wert in einen Integer konvertiert werden muss. Als Beispiel soll gezeigt werden, wie man den dritten Eintrag einer Listbox markieren kann:

```
SendMessage(hwndListbox, LB_SETSEL, WPARAM(true), 2);
```

Die Prozedur "SelAll" des Beispielprogramms demonstriert das Markieren aller Einträge der Listbox. Anzumerken ist aber, dass "LB_SETSEL" nur für eine Listbox gilt, in der man mehrere Einträge gleichzeitig auswählen kann.
Für die typischen Listboxen, die nur die Auswahl eines einzigen Items erlauben, muss die Nachricht "LB_SETCURSEL" benutzt werden. Hierbei geben Sie den gewünschten Index im wParam an, während der lParam nicht genutzt wird und Null bleibt:

```
SendMessage(hwndSimpleLB, LB_SETCURSEL, 2, 0);
```

Mit dem wParam-Wert -1 können Sie hier die Auswahl aufheben.

1.6.6. Auf die Änderung der Auswahl einer Listbox reagieren

Hier benutzen wir wieder die Nachricht "WM_COMMAND". Im höherwertigen Wort von wParam befindet sich die Listbox-Nachricht "LBN_SELCHANGE", die ausgelöst wird wenn die Auswahl des Listenfeldes ändert. (Andere Benachrichtigungscodes finden Sie in der Hilfe und im MSDN.) Im niederwertigen Wort von wParam befindet sich der Bezeichner des betreffenden Listenfeldes.

```
WM_COMMAND:
  case hiword(wParam) of
    LBN_SELCHANGE:
      case LoWord(wParam) of
        IDC_LB:
          begin
          end;
      end;
  end;
```

Zu Demonstrationszwecken benutze ich hier einmal "LB_GETSEL" um den Status eines Eintrages abzufragen. Die Alternative wäre wieder "".

```
Items := SendMessage(hwndListbox, LB_GETCOUNT, 0, 0);
for i := 0 to Items do
  if SendMessage(hwndListbox, LB_GETSEL, i, 0) > 0 then
    begin
      Inc(SelItems);
      wvsprintf(buffer1, 'davon markiert: %d', PChar(@SelItems));
      lstrcpy(buffer, buffer1);
    end;
```

1.6.7. Drag-Listboxen

1.6.7.1. Vorbemerkung

Als "DragListbox" bezeichnet man eine Listbox, deren Einträge mit der Maus verschoben werden können (to drag). Weil dabei Funktionen der so genannten Common Controls benutzt werden, ist es erforderlich, die Unit "`CommCtrl.pas`" einzubinden und den Befehl "`InitCommonControls`" (bzw. "`InitCommonControlsEx`") im Programm aufzurufen.

Ich habe die Demo als Konsolenanwendung definiert, `{$APPTYPE CONSOLE}`, um das so zusätzlich erzeugte Konsolenfenster zur Ausgabe von Text mittels "`writeln`" nutzen zu können. Genauer gesagt: ich gebe in diesem Konsolenfenster die beim Verschieben eines Eintrages mit der Maus erzeugten Nachrichten aus, um den Vorgang des Verschiebens zu verdeutlichen. Diese Zeilen habe ich mit dem Kommentar "DEBUG" gekennzeichnet.

1.6.7.2. Eine Listbox mutiert ...

Eine DragListbox ist nur eine etwas modifizierte normale Listbox. Letztendlich bringen wir ihr nur bei, Einträge per Drag&Drop verschieben zu können. Diese Modifizierung wird ganz einfach durch einen Aufruf der API-Funktion "`MakeDragList`" erreicht. Übergeben wird einfach das Handle der zu modifizierenden Listbox:

```
MakeDragList(GetDlgItem(HDlg,IDC_LISTBOX));
```

Der `bool`-Rückgabewert kann zur Kontrolle dienen. Sollte die Listbox nicht modifiziert werden können, sollten Sie das Programm beenden, bzw. zumindest alle Funktionen, die irgendwie von der Modifizierung abhängig sind, deaktivieren.

MakeDragList-Definition

```
BOOL MakeDragList(
  HWND hLB
);
```

Zusätzlich muss man noch eine Nachricht bei Windows registrieren, die von Windows generiert wird, wenn es zu einer Drag&Drop-Operation in der Listbox kommt:

```
DL_MESSAGE := RegisterWindowMessage(DRAGLISTMSGSTRING);
```

"`DL_MESSAGE`" ist unsere Nachricht, und hinter `DRAGLISTMSGSTRING` verbirgt sich eine vom System vorgegebene Stringkonstante.
Die Funktion "`RegisterWindowMessage`" erzeugt eine systemweit einzigartige Nachricht. Das bedeutet, ruft ein anderes Fenster die Funktion mit der gleichen Stringkonstante auf, dann wird kein neuer Wert erzeugt, sondern die Funktion liefert den zuvor registrierten numerischen Wert zurück.

Auf diese Weise könnten Sie Kontakt mit anderen Programmen aufnehmen, die solche Nachrichten im System registrieren. Aber das ist hier nicht das Thema ... :o)

RegisterWindowMessage-Definition

```
UINT RegisterWindowMessage(
  LPCTSTR lpString
);
```

1.6.7.3. Nachrichten der DragListbox

Die DragListbox generiert vier Nachrichten, die wir als Reaktion auf das "Ziehen" nutzen können:

Wert	Bedeutung
DL_BEGINDRAG	wird generiert, wenn der "Zieh"-Vorgang begonnen wird
DL_DRAGGING	wird während des "Ziehens" erzeugt

DL_CANCELDRAG	wird beim Abbrechen des Vorgangs erzeugt (ausgelöst durch ESC)
DL_DROPPED	wird beim Beenden des "Zieh"-Vorgangs ausgelöst

Da es sich aber bei unserer zuvor registrierten Drag-Nachricht (DL_MESSAGE) um eine Variable handelt, können wir die o.g. Benachrichtigungen nicht wie sonst üblich auswerten. Wir müssen das daher im **else**-Teil von **case** machen. Dieses Prinzip wird uns beim Suchen- und Ersetzendialog des Standarddialoge-Tutorials noch einmal begegnen

```
else
  begin
    result := FALSE;
    // message handling of the drag listbox comes here
    if DL_MESSAGE <> 0 then // DL_MESSAGE cannot be WM_NULL (=0)!
      if uMsg = DL_MESSAGE then
```

Welche Nachricht nun generiert wurde, sagt uns das `PDRAGLISTINFO`-Record, das uns im `lParam` übermittelt wird. Dabei ist für uns die Membervariable `uNotification` von Interesse, die die o.g. Benachrichtigungen der DragListbox enthält

```
        case PDRAGLISTINFO(lParam)^.uNotification of
          DL_BEGINDRAG:
            { ... }
          DL_DRAGGING:
            { ... }
          DL_CANCELDRAG:
            { ... }
          DL_DROPPED:
            { ... }
        end; // case (PDRAGLISTINFO(lParam)^.uNotification of)
  end; // else (case uMsg of)
```

DRAGLISTINFO-Definition

```
typedef struct {
    UINT uNotification;
    HWND hWnd;
    POINT ptCursor;
} DRAGLISTINFO
```

1.6.7.4. Die Funktion "DrawInsert"

Die Funktion "`DrawInsert`" dient dazu, den Vorgang zu visualisieren und dem Benutzer eine Orientierung zu geben, wo er sich gerade befindet. Dazu zeichnet sie einen Pfeil links neben die DragListbox auf das Elternfenster. Das folgende kleine Beispiel ermittelt zunächst das Item der Listbox, auf dem wir uns befinden:

```
ItemIdxDragging:= LBItemFromPt(GetDlgItem(hDlg, IDC_LISTBOX),
  PDRAGLISTINFO(lParam)^.ptCursor, TRUE);
```

Damit können wir den Pfeil anzeigen lassen

```
DrawInsert(hDlg, PDRAGLISTINFO(lParam)^.hWnd, ItemIdxDragging);
```

Die ersten beiden Paramter geben jeweils das Handle des Elternfensters und der Listbox an. Der letzte Parameter ist der Index des Items, auf dem wir uns gerade im Moment des "Ziehens" befinden.
Um die Markierung wieder zurückzusetzen bzw. verschwinden zu lassen, wird als letzter Paramter einfach `-1` übergeben:

```
DrawInsert(hDlg, PDRAGLISTINFO(lParam)^.hWnd, -1);
```

DrawInsert-Definition

```
void DrawInsert(
  HWND handParent,
  HWND hLB,
  int nItem
);
```

1.6.7.5. Was noch wichtig wäre...

Ich möchte Ihr Augenmerk an dieser Stelle speziell auf einen Punkt lenken (der Rest der Programmlogik kann anhand der ausführlich kommentierten Demo nachvollzogen werden)

```
// return the message result explicitly, otherwise we would not
// recieve DL_DRAGGING
SetWindowLong(hDlg, DWL_MSGRESULT, Integer(TRUE));

// message handled
Result := True;
```

Wie der Kommentar schon sagt, ist es wichtig, dass wir dem System mitteilen, dass wir die "DL_BEGINDRAG"-Nachricht behandelt haben. Täten wir dies nicht, würde uns Windows keine "DL_DRAGGING"-Nachrichten schicken, weil es nicht weiß, ob der "Zieh"-Vorgang begonnen hat oder nicht. Und im Zweifelsfall bleibt Windows eben stumm - sicher ist sicher. Da ich auch hier wieder aus Bequemlichkeit zu einer Dialogressource gegriffen habe, erfolgt die Benachrichtigung durch den Aufruf der Funktion "SetWindowLong" mit dem Index-Parameter DWL_MSGRESULT. Das PSDK bemerkt dazu (frei übersetzt):

PSDK
Wird "SetWindowLong" mit dem Index DWL_MSGRESULT benutzt, um den Rückgabewert einer Dialog-Prozedur zu setzen, sollte danach auch noch zusätzlich **true** zurückgegeben werden, da sonst der übergebene Rückgabewert wieder überschrieben werden könnte.

1.7. Arbeiten mit der Combobox

1.7.1. Die Combobox erzeugen

Die Combobox ist ohne VCL ein bisschen gewöhnungsbedürftig - aber machbar. Dieses Tutorial zeigt, wie man Eintrag hinzufügt, löscht und ausgewählte Einträge ausliest und anzeigt.

Das Element wird mit "CreateWindowEx" erzeugt. Wichtig sind hierbei aber zwei Dinge:
Die angegebene Höhe ist die Höhe des Editfeldes inklusive der ausgeklappten Liste! Soll in der Liste ein Scrollbalken erscheinen, muss zusätzlich der Stil WS_VSCROLL benutzt werden!

Die wichtigsten Combobox-Stile (weitere finden Sie in der Hilfe und im MSDN):

Wert	Bedeutung
CBS_AUTOHSCROLL	Scrollt Text automatisch nach rechts, wenn der Anwender ein Zeichen am Ende der Zeile eingibt. Ohne diesen Stil kann die Textlänge nur der Breite der Eingabezeile entsprechen.
CBS_DISABLENOSCROLL	Enthält die Liste nicht genügend Items zum Scrollen, wird der Scrollbalken deaktiviert dargestellt. Ohne diesen Stil wird der Scrollbalken in diesem Fall versteckt.
CBS_LOWERCASE, CBS_UPPERCASE	Mit diesen Stilattributen lässt sich der Text in Klein-, bzw. Großbuchstaben konvertieren. Beide Attribute schließen sich aber gegenseitig aus, deshalb bitte nur einen verwenden.
CBS_SORT	Sortiert die Strings in der Liste alfabetisch.

Oder als Auszug aus dem Beispielprogramm:

```
hCB := CreateWindowEx(0, 'COMBOBOX', '', WS_CHILD or WS_VISIBLE or
  CBS_AUTOHSCROLL or CBS_DROPDOWN or CBS_SORT or WS_VSCROLL, 10, 10, 150,
  150, hWnd, IDC_CB, hInstance, nil);
```

1.7.2. Der Combobox einen Eintrag hinzufügen

Es gibt zwei Möglichkeiten zum Hinzufügen von Einträgen: "CB_ADDSTRING" fügt einen String alphabetisch in die Liste ein (sofern das entsprechende Stilattribut gesetzt ist) oder hängt den String an die Liste an:

```
buffer := 'Peter';
SendMessage(hCB, CB_ADDSTRING, 0, Integer(@buffer));
```

"CB_INSERTSTRING" hingegen ignoriert die Sortierung; hier kann man einen String gezielt an einer bestimmten Position einfügen:

```
buffer := 'Luckie';
SendMessage(hCB, CB_INSERTSTRING, {Index ->} 3, Integer(@buffer));
```

Beide Nachrichten haben gemeinsam, dass sie als zweiten Parameter einen Zeiger auf den Textpuffer erwarten. Aber während bei "CB_INSERTSTRING" der erste Parameter der Index ist, an dem der Text eingefügt werden soll, muss dieser Wert bei "CB_ADDSTRING" Null sein.

1.7.3. Einen Eintrag aus der Combobox entfernen

Die Vorgehensweise beim Löschen eines Eintrages entspricht vom Prinzip her der Listbox. Zuerst sollte man mit der Nachricht "CB_GETCURSEL" ermitteln, welchen Index der ausgewählte Eintrag hat bevor man ihn mit "CB_DELETESTRING" entfernen kann. Der Auszug aus dem Beispielprogramm verdeutlicht es:

```
IDC_DEL:
  begin
    {Index holen}
    i := SendMessage(hCB, CB_GETCURSEL, i, 0);
    if(i = CB_ERR) then exit;

    {Item löschen}
    Sendmessage(hCB, CB_DELETESTRING, i, 0);
  end;
```

Um den gesamten Inhalt der Combobox zu löschen, verwenden Sie die Nachricht "CB_RESETCONTENT". Beide Parameter müssen hier Null sein:

```
IDC_DELALL:
  SendMessage(hCB, CB_RESETCONTENT, 0, 0);
```

1.7.4. Einen Eintrag aus der Combobox auslesen

Wenn Sie sich das Beispielprogramm anschauen, dann werden Sie bemerken, dass Ihnen in einem Label oben rechts der jeweils aktuell ausgewählte Eintrag der Combobox angezeigt wird. Dazu wird die Nachricht "CBN_SELCHANGE" bearbeitet. Sie ist ein Zeichen, dass sich die Auswahl der Combobox geändert hat. Mit der Nachricht "CB_GETLBTEXT" kann man dann den Text auslesen. Vorher ist natürlich wieder der Index des markierten Eintrags zu erfragen. Der Auszug zeigt Ihnen wie es geht:

```
WM_COMMAND:
  case hiword(wParam) of
    {Combobox-Auswahl geändert}
    CBN_SELCHANGE:
      case loword(wParam) of
        IDC_CB:
          begin
            {Index holen}
            i := SendMessage(hCB, CB_GETCURSEL, i, 0);
            if(i <> CB_ERR) then
              begin
                {Label aktualisieren}
                Sendmessage (hCB, CB_GETLBTEXT, i, Integer(@buffer));
                SendMessage (hSelItem, WM_SETTEXT, 0, Integer(@buffer));

                {Löschen-Button aktivieren}
                EnableWindow(hDel,true);
              end;
          end;
      end;
  end;
```

1.7.5. Die Anzahl der Einträge in der Combobox ermitteln

Um die Anzahl der Einträge bei einer Combobox zu ermitteln, benutzen wir die Nachricht "CB_GETCOUNT". Rückgabeergebnis ist der entsprechende Wert, bzw. CB_ERR im Fehlerfall.

```
i := SendMessage(hCB, CB_GETCOUNT, 0, 0);
wvsprintf(buffer, '%d', PChar(@i));
lstrcat(buffer, ' Einträge in der Combobox');
```

1.7.6. Eine ComboBoxEx erzeugen

Eine ComboBoxEx ist eine erweiterte Form der ComboBox, die eine eingebaute Unterstützung für Grafiken besitzt. Allerdings ist der Stil `CBS_OWNERDRAWVARIABLE` nicht möglich, weil die Unterstützung in Form von Imagelisten realisiert wurde. Das bedeutet, Sie können mehrere Grafiken in diese Liste laden und müssen sich nicht selbst um das Zeichnen der Bilder kümmern. Allerdings haben diese Grafiken dann immer die selbe Größe. Ansonsten verhält sich die ComboBoxEx wie eine normale ComboBox.

Allerdings handelt es sich hier um ein so genanntes "Common Control". Es ist daher sehr wichtig, dass die ComboBoxEx zwingend mit "`InitCommonControlsEx`" initialisiert werden muss, sonst sehen Sie sie erst gar nicht

```
var
  iccex : TInitCommonControlsEx =
    (dwSize:sizeof(iccex);
     dwICC:ICC_USEREX_CLASSES;);
begin
  InitCommonControlsEx(iccex);

  { ... }
end.
```

Sie sehen sie auch nicht, wenn Sie nur "`InitCommonControls`" aufrufen, wie Sie das evtl. bei einigen anderen "Common Controls" noch sehen werden.

1.7.6.1. Das Control erzeugen

Wenn Sie die "Common Controls" und damit die ComboBoxEx initialisiert haben, dann können Sie sie wie jedes andere Control erzeugen, wobei als Klassenname `WC_COMBOBOXEX` zu benutzen ist:

```
hCBEx := CreateWindowEx(0,WC_COMBOBOXEX,nil,WS_BORDER or WS_CHILD or
  WS_VISIBLE or CBS_DROPDOWN,10,10,300,150,wnd,IDC_CBEX,hInstance,nil);
```

Erweiterte Stilattribute

Wert	Bedeutung
CBES_EX_CASESENSITIVE	Die Suche nach Text in der ComboBoxEx ist abhängig von der Schreibweise der Strings.
CBES_EX_NOEDITIMAGE	Das Eingabefeld und die Liste der ComboBoxEx zeigen keine Grafiken an
CBES_EX_NOEDITIMAGEINDENT	Das Eingabefeld und die Liste der ComboBoxEx zeigen keine Grafiken an

(weitere Stile finden Sie im PSDK)

Wenn Sie einen der erweiterten Stile setzen wollen, dann benötigen Sie die Nachricht "`CBEM_SETEXTENDEDSTYLE`", wobei Sie den `wParam` auf Null setzen sollten. Das bewirkt, dass alle Flags, die Sie im `lParam` angeben, berücksichtigt werden. bspw.

```
SendMessage(hCBEx,CBEM_SETEXTENDEDSTYLE,0,CBES_EX_NOEDITIMAGEINDENT);
```

Im Gegensatz dazu wäre im folgenden Aufruf das Flag `CBES_EX_CASESENSITIVE` wirkungslos, weil der `wParam` lediglich `CBES_EX_NOEDITIMAGE` enthält und damit das zweite Flag im `lParam` ignoriert

```
SendMessage(hCBEx, CBEM_SETEXTENDEDSTYLE, CBES_EX_NOEDITIMAGE,
  CBES_EX_NOEDITIMAGE or CBES_EX_CASESENSITIVE);
```

1.7.6.2. Die Imageliste zuweisen

Wie bereits erwähnt enthält die ComboBoxEx eine eingebaute Unterstützung für Grafiken in Form von Imagelisten. Das heißt, wenn Sie die Grafiken nicht gerade zur Laufzeit erzeugen, sondern bspw. Bilder aus den Ressourcen in der ComboBoxEx anzeigen lassen wollen, dann müssen Sie lediglich eine Imageliste erzeugen und die notwendigen Grafiken laden.

Leider muss ich an der Stelle ein bisschen vorgreifen. Zum einen werden uns Imagelisten noch in späteren Beiträgen begegnen (bei der List-View etwa), und zum anderen benutzen wir kurzerhand die Grafik aus dem Toolbar-Beitrag.
Gehen wir also davon aus, dass die besagte Grafik Teil der Programmressourcen ist und die ID `100` hat. Dann laden wir sie zuerst

```
hbmp := LoadBitmap(hInstance,MAKEINTRESOURCE(100));
```

Wenn das funktioniert hat, dann ist der Rückgabewert ungleich Null, und wir können die Imageliste mit den Abmessungen von 16x16 Pixel erzeugen

```
if(hbmp <> 0) then begin
  hImgList := ImageList_Create(16,16,ILC_COLOR,0,1);
```

Nun laden wir die Grafik, indem wir lediglich den Befehl "`ImageList_Add`" aufrufen. Das Besondere dabei ist, dass unsere Grafik eigentlich mehrere Einzelbilder enthält. Durch die Festlegung der Imageliste auf 16 Pixel Breite wird die Grafik daher automatisch in die Einzelbilder "aufgetrennt". Sie müssen daher nur darauf achten, dass alle Einzelbilder die selbe Breite haben

```
  ImageList_Add(hImgList,hbmp,0);
```

Da wir die geladene Grafik nicht mehr benötigen, geben wir sie frei

```
  DeleteObject(hbmp);
```

und übergeben dann die Imageliste mit der Nachricht "`CBEM_SETIMAGELIST`" an die ComboBoxEx

```
  SendMessage(hCBEx,CBEM_SETIMAGELIST,0,LPARAM(hImgList));
end;
```

Die Imageliste freigeben

Bevor Sie das Programm beenden, sollten Sie die Imageliste mit dem Befehl "ImageList_Destroy" wieder freigeben, wobei Sie als Parameter das Handle der Liste benutzen:

```
ImageList_Destroy(hImgList);
```

1.7.6.3. Items einfügen

Um ein Item einzufügen, benötigen wir das Record `TComboBoxExItem`. Zuerst legen wir fest, welche Membervariablen des Records gültig sein sollen. Dazu dient die `mask`-Variable, der wir die notwendigen Flags übergeben. Wir wollen einen Text in der ComboBoxEx sehen, der Text soll u.U. eingerückt sein, und er soll eine Grafik anzeigen. Die Festlegung der Attribute lautet daher also

```
cbei.mask := CBEIF_TEXT or CBEIF_INDENT or CBEIF_IMAGE or
  CBEIF_SELECTEDIMAGE;
```

Dann geben wir den Index des Items an. Im Beispielprogramm wird hier eine Schleife durchlaufen, und der jeweilige Schleifenwert wird als Index übergeben. Wenn Sie stattdessen jedes neue Item generell an das Ende anhängen wollen, dann benutzen Sie den Wert -1

```
cbei.iItem          := -1;
```

Danach geben wir den gewünschten Text und dessen Länge an. Im Beispielprogramm kommen diese Angaben aus einem Array, dessen Elemente durch die Schleife nacheinander abgearbeitet werden. Hier soll ein Wert aus dem Array das Prinzip verdeutlichen:

```
cbei.pszText        := 'Peter';
cbei.cchTextMax     := lstrlen('Peter');
```

Wenn Sie den Text einrücken wollen, dann benutzen Sie die `iIndent`-Membervariable und weisen Sie ihr einen Wert zu. Zu beachten ist, dass der angegebene Wert mit 10 multipliziert wird, d.h. Eins entspricht also einer Einrückung von 10 Pixel, Zwei von 20 Pixel, usw.

```
cbei.iIndent        := 1;
```

Bleiben noch die Grafiken. In der Auswahlliste soll vor jedem Item ein Bild aus der Imageliste zu sehen sein. Darum müssen wir dessen Index natürlich angeben:

```
cbei.iImage         := ItemInfo[i].iImage;
```

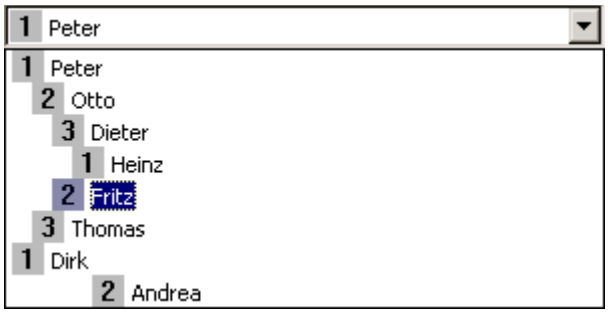
Aber auch bei der Auswahl eines Items soll im Eingabefeld der ComboBoxEx die Grafik zu sehen sein. Aus dem Grund haben wir das Flag `CBEIF_SELECTEDIMAGE` benutzt, was bedeutet, das wir den Index des gewünschten Bildes an die `iSelectedImage`-Variable übergeben müssen:

```
cbei.iSelectedImage := ItemInfo[i].iImage;
```

Mit Hilfe der Nachricht "`CBEM_INSERTITEM`" fügen wir das Item dann ein

```
SendMessage(hCBEx,CBEM_INSERTITEM,0,LPARAM(@cbei));
```

und erhalten beispielsweise das folgende Ergebnis:



Und damit sehen Sie, dass Sie sich tatsächlich um nichts weiter kümmern müssen. Die Höhe der ComboBoxEx richtet sich übrigens nach der Höhe der Grafiken. Das Beispielprogramm enthält einen Compilerschalter (`THE_BIG_PICTURE`), mit dem Sie eine 32x32-Bitmap laden lassen können. Die ComboBoxEx ist dann entsprechend höher.

TcomboBoxExItem-Definition

```
typedef struct {
    UINT mask;          // legt die gültigen Membervariablen fest
    INT_PTR iItem;      // Item-Index, bzw. -1
    LPTSTR pszText;     // Itemtext
    int cchTextMax;     // Länge des Textes
    int iImage;         // Index der Grafik für die Auswahlliste
    int iSelectedImage; // Index der Grafik für das Eingabefeld
    int iOverlay;
    int iIndent;        // Einrückung
    LPARAM lParam;
} COMBOBOXEXITEM
```

1.8. Ein Menü hinzufügen

1.8.1. Das Menü erzeugen

Dieses Tutorial erklärt das Erzeugen und den Einsatz eines Menüs durch API-Aufrufe.

Um ein Hauptmenü zu erzeugen, benutzen wir zuerst die Funktion "CreateMenu". Der Rückgabewert dieser Funktion wäre dann das Handle unseres Menüs, dem wir mit "AppendMenu" Einträge zuordnen können. Das bezieht sich allerdings nur auf die Menüleiste unterhalb der Titelzeile - etwa mit Items wie "Datei", "Bearbeiten" usw.

```
hMenu := CreateMenu;
```

Um auch diese Einträge mit Leben zu füllen, benötigen wir ein Untermenü, das wir mit der Funktion "CreatePopupMenu" erzeugen. Auch hier erhalten wir ein Handle zurück, so dass wir wieder mit "AppendMenu" die eigentlichen Einträge anhängen können:

```
hSubMenu := CreatePopupMenu;

{ Menüs mit Einträgen füllen }
AppendMenu(hSubMenu, MF_STRING, IDM_ITEM1, 'Item&1');
AppendMenu(hSubMenu, MF_STRING, IDM_ITEM2, 'Item&2');
AppendMenu(hSubMenu, MF_SEPARATOR, 0, nil);
AppendMenu(hSubMenu, MF_STRING, IDM_ITEM3, 'Item&3');
```

Das so erstellte Untermenü wird dann dem Hauptmenü zugeordnet, so das letztlich die gewünschte Struktur entsteht:

```
AppendMenu(hMenu, MF_STRING or MF_POPUP, hSubMenu, 'Menü&1');
```

Und natürlich müssen wir das Menü an das Fenster übergeben, damit wir es auch sehen:

```
SetMenu(hWnd, hMenu);
```

Genaue Informationen zu den Funktionen entnehmen Sie bitte dem MSDN oder der Hilfe. Ich will nur kurz auf ein paar Stilattribute eingehen, die aber ebenfalls in den Hilfen erklärt werden:

Wert	Bedeutung
MF_CHECKED, MF_UNCHECKED	Setzt oder entfernt den Haken vor einem Menüeintrag.
MF_SEPARATOR	Dieser Eintrag bezeichnet einen Trennstrich. Als nächste ID kann Null, und als Menütext **nil** angegeben werden, da dieser Eintrag in der Fensterfunktion nicht abgefragt wird.
MF_DISABLED, MF_GRAYED	Beide Attribute deaktivieren den Menüeintrag, aber nur der zweite stellt ihn auch grau dar.
MF_ENABLED	Aktiviert einen Menüeintrag wieder.
MF_POPUP	Anstelle der Item-ID wird hier das Handle des Untermenüs angegeben.

1.8.2. Auf die Itemauswahl von Menüs reagieren

Menüklicks werden wie Buttonklicks behandelt. Es muss innerhalb von "WM_COMMAND" die Nachricht "BN_CLICKED" behandelt werden, wobei natürlich die IDs der Menüeinträge nicht zu vergessen sind:

```
WM_COMMAND:
  if hiword(wParam) = BN_CLICKED then
    case loword(wParam) of
      IDM_ITEM1:
        begin
          buffer := 'Item1';
          ShowMsgBox(hWnd, buffer);
        end;
    end;
```

usw.

1.8.3. Menü-Einträge aktivieren und deaktivieren

Zum Aktivieren und Deaktivieren von Menüitems bietet das API die Funktion "`EnableMenuItem`". Dieser Funktion übergeben Sie zuerst das Handle des Menüs. Der zweite Parameter bezeichnet die Item-ID des Eintrags, der aktiviert oder deaktiviert werden soll. Und der letzte Parameter gibt dann genau diesen Status an. Die beiden am häufigsten benutzten Flags werden wohl `MF_ENABLED` und `MF_GRAYED` sein, über weitere gibt die Hilfe wie gewohnt Auskunft. Im Code sieht das dann so aus:

```
WM_COMMAND:
  if hiword(wParam) = BN_CLICKED then
    case loword(wParam) of
      IDC_ENABLE:
        EnableMenuItem(hMenu, IDM_ITEM1, MF_ENABLED);
      IDC_DISABLE:
        EnableMenuItem(hMenu, IDM_ITEM1, MF_GRAYED);
    end;
```

EnableMenuItem-Definition

```
BOOL EnableMenuItem(
  HMENU hMenu,          // handle to menu
  UINT uIDEnableItem,   // menu item to enable, disable, or gray
  UINT uEnable          // menu item flags
);
```

1.8.4. Informationen über einen Menü-Eintrag

Möchte man den Status eines Menüitems herausfinden, bietet das API dafür die Funktion "`GetMenuItemInfo`". Diese Funktion benötigt u.a. ein Record vom Typ `TMenuItemInfo`. Es muss zuerst initialisiert werden. Außerdem müssen wir in der `fMask`-Variablen angeben, an welchen Informationen wir interessiert sind. In diesem Fall ist es der Status des Eintrags. (Der Parameter kann weitere Angaben enthalten, die dann entsprechend **OR**-verknüpft werden müssten.)

```
MenuItemInfo.cbSize := sizeof(TMenuItemInfo);
MenuItemInfo.fMask  := MIIM_STATE;
```

TMenuItemInfo-Definition

```
typedef struct tagMENUITEMINFO {
    UINT    cbSize;           // Größe des Records
    UINT    fMask;            // Maske zur Auswahl des Typs
    UINT    fType;
    UINT    fState;
    UINT    wID;
    HMENU   hSubMenu;
    HBITMAP hbmpChecked;
    HBITMAP hbmpUnchecked;
    DWORD   dwItemData;
    LPTSTR  dwTypeData;
    UINT    cch;
} MENUITEMINFO, FAR *LPMENUITEMINFO;
```

Nun können wir die Funktion "GetMenuItemInfo" aufrufen, wobei wir das Ergebnis gleich an eine Bool-Variable weiterleiten:

```
isEnabled := (GetMenuItemInfo(hMenu,IDM_ITEM1,false,MenuItemInfo)) and
  (MenuItemInfo.fState and MFS_GRAYED = 0);
```

GetMenuItemInfo-Definition

```
BOOL GetMenuItemInfo(
  HMENU hMenu,              // Menü-Handle
  UINT uItem,               // Menüitem-ID
  BOOL fByPosition,
  LPMENUITEMINFO lpmii      // unsere "TMenuItemInfo"-Record
);
```

1.8.5. GetMenuState

Wenn Sie nur an den Statusinformationen eines Eintrags und nicht an weitergehenden Informationen interessiert sind, dann geht das auch mit der "GetMenuState"-Funktion. Unser Beispiel von eben reduziert sich damit auf diese eine Zeile:

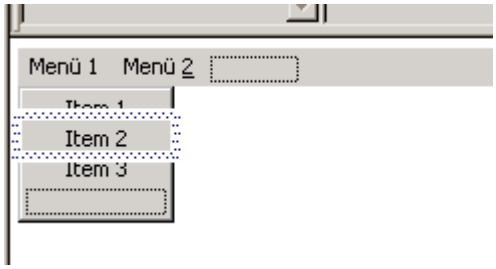
```
isEnabled := GetMenuState(hMenu,IDM_ITEM1,MF_BYCOMMAND) <> MF_GRAYED;
```

GetMenuState-Definition

```
UINT GetMenuState(
    HMENU hMenu,              // Menü-Handle
    UINT uId,                 // Menüitem-ID, oder Menüitem-Index
    UINT uFlags               // MF_BYCOMMAND  (uId = Menü-ID)
                              // MY_BYPOSITION (uId = Item-Index)
);
```

1.8.6. Menüs aus der Ressource laden

Nun muss man das Menü natürlich nicht erst beim Start des Programms "zusammenbauen". Man kann das Menü auch aus der Ressource des Programms zu laden. Dazu benötigen wir zuerst einen Ressourcen-Editor, der die Gestaltung von Menüs erlaubt. Beim Thema Dialoge gehen wir etwas ausführlicher auf den VisualStudio-Editor von Microsoft ein, deshalb sei hier nur gesagt, dass ich eben diesen Editor für das Menü benutzt habe:



Im Quelltext des Ressourcen-Skriptes sieht unser Menü dann so aus:

```
// Menu
200 MENU DISCARDABLE
BEGIN
    POPUP "Menü 1"
    BEGIN
        MENUITEM "Item 1",                      1
        MENUITEM "Item 2",                      2
        MENUITEM "Item 3",                      3
    END
    POPUP "Menü &2"
    BEGIN
        MENUITEM "Item 4",                      4
        MENUITEM "Item 5",                      5
    END
END
```

Dieses Skript wandeln wir mit einem Ressourcen-Compiler, wie "brcc32" von Borland, in eine Ressourcendatei um und binden sie in unser Programm ein. Um das Menü nutzen zu können, benötigen wir den API-Befehl "LoadMenu", der als Parameter neben der Programminstanz auch die ID der Menüressource erwartet:

```
hMenu := LoadMenu(hInstance,MAKEINTRESOURCE(200));
```

Es gibt zwei Varianten zum Laden. Zum einen können wir das Menü im "WM_CREATE"-Teil der Nachrichtenfunktion laden lassen, müssen es dann aber noch an das Fenster "übergeben". Das passiert wie gehabt mit "SetMenu".
Als zweite Möglichkeit können wir den Befehl "LoadMenu" als Parameter von "CreateWindowEx" angeben:

```
CreateWindowEx(0, ClassName, AppName, WS_CAPTION or WS_VISIBLE or WS_SYSMENU
  or WS_MINIMIZEBOX or WS_MAXIMIZEBOX or WS_SIZEBOX, CW_USEDEFAULT,
  CW_USEDEFAULT, WindowWidth, WindowHeight, 0,
  LoadMenu(hInstance,MAKEINTRESOURCE(200)), // <-- Menü laden
  hInstance, nil);
```

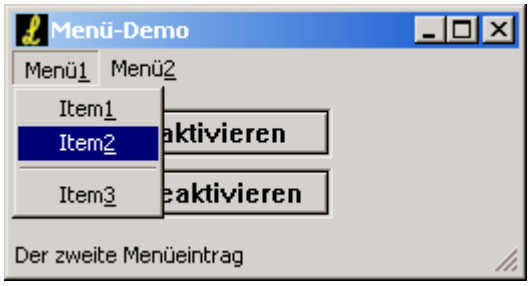
LoadMenu-Definition zeigen

```
HMENU LoadMenu(
  HINSTANCE hInstance,  // handle to module
  LPCTSTR lpMenuName    // menu name or resource identifier
);
```

1.8.7. Hilfe zum Menüpunkt in der Statuszeile anzeigen

In vielen Programmen sehen Sie eine kurze Erklärung in der Statuszeile, wenn Sie mit dem Mauszeiger über einem Menüeintrag sind:

Das System stellt Ihnen dazu die Funktion "MenuHelp" zur Verfügung, die Sie am besten bei der Bearbeitung der



Nachricht "WM_MENUSELECT" aufrufen.

Wie der Name schon sagt, wird die Nachricht bei der Auswahl eines Menüeintrags ausgelöst. Der lParam enthält dabei das Handle des Menüs. Das niederwertige Word des wParam gibt das Menüitem oder den Submenü-Index an. Das höherwertige Word spezifiziert Menü-Flags, die ausgewertet werden können, beispielsweise MF_BITMAP, MF_GRAYED, usw.

Im Beispielprogramm wird so geprüft, ob der ausgewählte Menüeintrag ein Separator oder ein Popupmenü ist. In diesem Fall wird an die Statuszeile die Nachricht "SB_SIMPLE" gesendet, was sozusagen ein "Neuzeichnen" der Statuszeile zur Folge hat.

```
if(bool(HIWORD(wParam) and MF_POPUP)) or
  (bool(HIWORD(wParam) and MF_SEPARATOR)) or
  (HIWORD(wParam) = $FFFF) then SendMessage(hStatus,SB_SIMPLE,0,0)
```

Andernfalls rufen wir die schon genannte "MenuHelp"-Funktion auf, der wir die Nachricht, den wParam- und den lParam-Wert, sowie das Menü- und Statuszeilen-Handle und die Anwendungsinstanz übergeben müssen:

```
MenuHelp(uMsg,wParam,lParam,HMENU(lParam),hInstance,
  hStatus,@dummy);
```

Die Reihenfolge der Werte ist aus dieser Beispielzeile ja ersichtlich. Erwähnenswert ist noch der letzte Parameter. Laut Microsoft soll dieser Parameter ein Array sein, das paarweise die ID einer String-Ressource und eines Menü-Handles beinhaltet. Dieses Array soll dann nach dem Handle durchsucht werden, worauf (im Erfolgsfall, natürlich!) die Funktion den passenden String aus den Ressourcen des Programms lädt und in der Statuszeile anzeigt. So steht es zumindest im PSDK.

Nun ist die Variable "dummy" allerdings so deklariert:

```
var
  dummy : uint = 0;
```

Das ist möglich, weil laut Definition der letzte Parameter ein Zeiger auf eine UINT-Variable ist. Und es sieht so aus, als würde der Wert dieser Variablen zum niederwertigen Wort von wParam addiert werden, um die gewünschte Ressourcen-ID zu bilden. Das lässt sich sogar nachprüfen, wenn Sie den Wert von "dummy" z.B. auf Eins ändern und das Beispiel ausprobieren. In dem Fall würde der erste Menüeintrag nun die Information des zweiten Eintrags anzeigen, und der letzte (Nr. 5) hätte überhaupt keine passende Ressource mehr.

Da unsere String-Ressourcen nun die selben IDs wie die Menüeinträge benutzen

```
STRINGTABLE DISCARDABLE
BEGIN
    1                       "Der erste Menüeintrag"
    2                       "Der zweite Menüeintrag"
    3                       "Der dritte Menüeintrag"
    4                       "Menüeintrag Nr. 4"
    5                       "Und natürlich - der fünfte Eintrag"
END
```

belassen wir den Wert von "dummy" auf Null, und alles ist in Butter ... :o)

Hinweis

Wenn Sie Ihr Menü z.B. mit dem Ressourcen-Editor des Visual Studio erstellen, haben Sie dort gleich die Möglichkeit, den passenden Text einzutragen. Dadurch wird automatisch die passende String-Ressource angelegt und gespeichert.

MenuHelp-Definition

```
void MenuHelp(
    UINT uMsg,          // WM_MENUSELECT oder WM_COMMAND
    WPARAM wParam,      // wParam der Nachricht
    LPARAM lParam,      // lParam der Nachricht
    HMENU hMainMenu,    // Handle des Menüs
    HINSTANCE hInst,    // Modulinstanz (für die String-Ressourcen)
    HWND hwndStatus,    // Handle der Statuszeile
    LPUINT lpwIDs       // Zeiger auf UINT-Variable
);
```

1.9. Dialoge aus Ressourcen aufrufen

1.9.1. Grundlagen

In diesem Tutorial geht es um die Dialoge aus den Programmressourcen. Bevor man immer neue Fenster erzeugt und aufruft, kann es hilfreicher sein diese vorgefertigten "Formulare" zu nutzen.

Es bleibt Ihnen überlassen, mit welchem Editor Sie Dialoge erstellen. Der Klassiker schlechthin ist wohl Borlands ResourceWorkshop, der - leider nicht sehr aktuell - bei Delphi 5 noch dabei ist. Ich persönlich bevorzuge den VisualStudio-Editor von Microsoft. Abhängig von der verwendeten VisualStudio-Version ist er recht aktuell und bietet daher auch Zugriff auf neuere Controls (etwa **TreeView** oder **ListView**), die dem Borland-Editor unbekannt sind. Wenn Sie den VS-Editor ebenfalls benutzen, dann möchte ich Sie auf ein paar Dinge aufmerksam machen:

1. Ein Phänomen, das mich am Anfang so manche Stunde gekostet hat, ist die Tatsache, dass die erzeugten Dialoge standardmäßig nicht angezeigt werden. Schuld daran ist der fehlende Haken vor der Option "Sichtbar" in den Eigenschaften des Dialogfeldes. Wenn Ihr Programm also läuft, Sie aber nichts sehen, dann dürfte diese fehlende Sichtbarkeit der Hauptgrund sein.
2. Beim VS-Editor haben Sie die Möglichkeit, die Ressourcen als Skript (*.rc) oder direkt als kompilierte Ressourcendatei (*.res) zu speichern. Um das Skript nutzen zu können, müssen Sie es erst mit einem Ressourcen-Compiler - z.B. dem "`bcc32.exe`" von Borland - kompilieren. Dieser Schritt entfällt bei der fertigen RES-Datei, dafür können Sie diese nicht mehr manuell nachbearbeiten - was beim Skript möglich ist, weil es sich im Prinzip nur um eine Textdatei mit bestimmter Syntax handelt.
3. Der VS-Editor erzeugt neben dem Skript auch noch eine "`resource.h`". Es kann zu einer Fehlermeldung kommen, wenn Sie versuchen das RC-Skript mit dem Ressourcen-Compiler von Borland zu kompilieren. Das liegt aber nicht an Inkompatibilitäten sondern einfach nur daran, dass der Compiler auf bestimmte Definitionen nicht zugreifen kann.
   In diesem Fall sollten Sie in Ihre "`autoexec.bat`" den Namen der VisualStudio-Batchdatei eintragen, die die Umgebungsvariablen anpasst. Standardmäßig heißt diese Datei "`VcVars32.bat`" und befindet sich im BIN-Ordner von VisualC++. Nach dem nächsten Neustart erzeugt Borlands Ressourcen-Compiler ohne weitere Probleme Ihre RES-Datei.
4. Die Abmessungen aller Elemente des VS-Editors entsprechen nicht dem Pixelmaß von Delphi. Darauf sollten Sie ganz besonders achten. Wenn Sie - wie von Delphi gewohnt - einem Element z.B. als Breite den Wert 300 zuordnen, dann sind das mehr als 300 Pixel.

Im folgenden Beispiel wollen wir diesen Dialog:

in unserem Programm anzeigen, der sich in einem Ressourcen-Skript ("`dialog.rc`") befindet. Ich sagte ja schon, dass es sich dabei nur um eine Textdatei mit bestimmter Syntax handelt, und deshalb sieht unser Dialog im Skript so aus:

```
// Dialog
100 DIALOGEX 0, 0, 162, 95
STYLE DS_3DLOOK | DS_NOFAILCREATE | DS_CENTER | WS_MINIMIZEBOX |
    WS_MAXIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Arbeiten mit Dialog-Ressourcen"
FONT 8, "MS Sans Serif"
BEGIN
    CTEXT           "Hello",101,38,5,85,20,SS_CENTERIMAGE | SS_SUNKEN
    CTEXT           "World",102,38,35,85,20,SS_CENTERIMAGE | SS_SUNKEN
    PUSHBUTTON      "&Beschriften",103,5,70,70,20,0,WS_EX_CLIENTEDGE
    PUSHBUTTON      "&Schließen",104,85,70,70,20,0,WS_EX_CLIENTEDGE
END
```

1.9.2. Nicht-modale Dialoge

Mit der Funktion "`CreateDialog`" erzeugen Sie einen nicht-modalen Dialog. Neben der Anwendungsinstanz übergeben Sie die ID der Dialog-Ressource, das Besitzerfenster (hier Null) und einen Zeiger auf die Nachrichtenfunktion. Wenn wir bei unserem Skriptbeispiel bleiben, dann würde der Aufruf so aussehen:

```
hDialog := CreateDialog(hInstance, MAKEINTRESOURCE(100), 0, @dlgfunc);
```

Laut MSDN nutzt diese Funktion aber auch nur "`CreateDialogParam`", die Sie also stattdessen aufrufen könnten. Neu ist hier nur der Initialisierungswert, den Sie nicht zwangsläufig nutzen müssen. Sie können hier aber einen Wert für `lParam` angeben, der dann an Ihre Dialog-Nachrichtenfunktion weitergegeben wird:

```
hDialog := CreateDialogParam(hInstance, MAKEINTRESOURCE(100), 0, @dlgfunc, 0);
```

CreateDialog-Definition

```
HWND CreateDialog(
  HINSTANCE hInstance,      // Anwendungsinstanz
  LPCTSTR lpTemplate,       // ID der Dialog-Ressource
  HWND hWndParent,          // Handle des Besitzerfensters
  DLGPROC lpDialogFunc      // Zeiger auf die Dialog-Nachrichtenfunktion
);
```

CreateDialogParam-Definition

```
HWND CreateDialogParam(
  HINSTANCE hInstance,      // Anwendungsinstanz
  LPCTSTR lpTemplateName,   // ID der Dialog-Ressource
  HWND hWndParent,          // Handle des Besitzerfensters
  DLGPROC lpDialogFunc,     // Zeiger auf die Dialog-Nachrichtenfunktion
  LPARAM dwInitParam        // Initialisierungswert
);
```

Wenn Sie eine dieser Funktionen benutzen, dann benötigen Sie auch eine Nachrichtenschleife, in der Sie ankommende Nachrichten abfangen. Allerdings unterscheidet sich die Dialog-Nachrichtenschleife durch den Aufruf der Funktion "`IsDialogMessage`". Weil diese Funktion das Übersetzen und Verteilen der Nachrichten selbst ausführt, darf sie nicht von "`TranslateMessage`" oder "`DispatchMessage`" bearbeitet werden.

```
while GetMessage(msg,0,0,0) do
  begin
    IsDialogMessage(hDialog,msg);
  end;
```

IsDialogMessage-Definition

```
BOOL IsDialogMessage(
  HWND hDlg,                // Dialog-Handle
  LPMSG lpMsg               // "TMsg"-Struktur
);
```

Hinweis

Wenn Sie in Ihrem Programm die Nachrichten "WM_USER" und "WM_USER + 1" benutzen, könnten u.U. Probleme auftreten, weil "IsDialogMessage" in einigen Fällen die Nachrichten "DM_GETDEFID" und "DM_SETDEFID" senden kann. Diese besitzen die selben numerischen Werte wie die o.g. Nachrichten, so dass es zu Überschneidungen und damit verbundenen Problemen kommt, wenn Ihr Programm z.B. eine Bearbeitung von "WM_USER + 1" vorsieht.

Wenn Sie sich für diese Form des Dialogaufrufs entscheiden, müssen Sie sich im Klaren sein, dass Sie die Verantwortung für die Verarbeitung aller Nachrichten haben. Das betrifft alle Controls, die Sie auf dem Dialog platziert haben usw. Oder wenn Sie z.B. Shortcuts aus den Ressourcen laden und bearbeiten wollen, dann bleibt Ihnen leider nichts anderes übrig, als diesen Weg zu wählen.

Nach dem Verlassen der Nachrichtenschleife sollten Sie auch das Dialog-Handle mit der Funktion "DestroyWindow" freigeben:

```
DestroyWindow(hDialog);
```

1.9.3. Modale Dialoge

Mit der Funktion "DialogBox" erzeugen Sie einen modalen Dialog:

```
DialogBox(hInstance, MAKEINTRESOURCE(100), 0, @dlgfunc)
```

Auch diese Funktion nutzt eigentlich die Fähigkeiten einer anderen - "DialogBoxParam" - die Sie ebenso aufrufen könnten. Die Parameter entsprechen der Funktion "CreateDialog" bzw. "CreateDialogParam". Der große Unterschied ist aber, dass Sie hier keine Nachrichtenschleife benötigen. Eine Nachrichtenfunktion schon, aber das Verteilen ankommender Nachrichten übernimmt der Dialogmanager des Systems.

DialogBox-Definition

```
int DialogBox(
  HINSTANCE hInstance,      // Anwendungsinstanz
  LPCTSTR lpTemplate,       // ID der Dialog-Ressource
  HWND hWndParent,          // Handle des Besitzerfensters
  DLGPROC lpDialogFunc      // Zeiger auf die Dialog-Nachrichtenfunktion
);
```

DialogBoxParam-Definition

```
int DialogBoxParam(
  HINSTANCE hInstance,      // Anwendungsinstanz
  LPCTSTR lpTemplateName,   // ID der Dialog-Ressource
  HWND hWndParent,          // Handle des Besitzerfensters
  DLGPROC lpDialogFunc,     // Zeiger auf die Dialog-Nachrichtenfunktion
  LPARAM dwInitParam        // Initialisierungswert
);
```

Und weil es in unserem Programm keine Nachrichtenschleife für den Dialog gibt, genügt zum Beenden in der Nachrichtenfunktion der Befehl "EndDialog".

```
case uMsg of
  WM_CLOSE:
    EndDialog(hDlg, 0);
end;
```

Dieser Befehl zerstört das Fenster aber nicht sofort. Es setzt stattdessen ein Flag und erlaubt so der Dialog-Prozedur die Kontrolle an das System zurückzugeben. Das System prüft das Flag, bevor es versucht die nächste Nachricht zu empfangen. Ist es gesetzt, dann beendet das System die (in dem Fall: interne) Nachrichtenschleife, zerstört das Fenster und benutzt den Wert von "nResult" als Rückgabewert für die Funktion, die den Dialog erstellt hat.

EndDialog-Definition

```
BOOL EndDialog(
  HWND hDlg,                // Dialogbox-Handle
  int nResult               // Rückgabewert
);
```

Das heißt auch, dass Sie kein Handle für Ihren Dialog benötigen. Da der Rückgabewert "nResult" entspricht (bzw. im Fehlerfall: -1) müssen Sie eigentlich nur prüfen, ob ein Fehler aufgetreten ist oder nicht, beispielsweise:

```
if(DialogBox(hInstance, MAKEINTRESOURCE(100), 0, @dlgfunc) = -1) then
  MessageBox(0,'Fehler beim Anzeigen des Dialogs','Dialog-Demo',MB_OK or
MB_ICONERROR);
```

1.9.4. Die Nachrichtenfunktion für Dialoge

Unsere Dialoge benötigen eine Funktion, in der wir auf ankommende Nachrichten reagieren können. Das Prinzip entspricht also der herkömmlichen "WndProc"-Funktion bei Fenstern, unterscheidet sich aber im Bool-Rückgabewert. Wenn die Funktion als Ergebnis **false** zurückliefert, ist das quasi das Zeichen, dass der Standard-Handler stattdessen aktiv werden soll. Wenn Sie selbst Nachrichten bearbeiten, dann sollten Sie unbedingt **true** zurückgeben, um die Abarbeitung durch den Standard-Handler zu vermeiden.

```
function dlgfunc(hDlg: hWnd; uMsg: dword; wParam: wParam; lParam: lParam): bool;
  stdcall;
```

Und noch einen Unterschied gibt es: Auf die Nachricht "WM_CREATE" werden Sie vergeblich warten. Das Gegenstück heißt hier "WM_INITDIALOG".

DialogProc-Definition

```
BOOL CALLBACK DialogProc(
  HWND hwndDlg,  // Dialog-Handle
  UINT uMsg,     // message
  WPARAM wParam, // first message parameter
  LPARAM lParam  // second message parameter
);
```

Beim Beenden des Dialogs kommt es nun darauf an, für welche Funktion Sie sich entschieden haben. Nutzen Sie "CreateDialog", dann müssen Sie den Dialog - wie bei der herkömmlichen Fensterfunktion - mit "PostQuitMessage(0)" beenden um die Nachrichtenschleife verlassen zu können:

```
WM_CLOSE:
  PostQuitMessage(0);
```

Nutzen Sie stattdessen "DialogBox", genügt der Aufruf des schon erwähnten Befehls "EndDialog":

```
WM_CLOSE:
  EndDialog(hDlg,0);
```

Im Beispielprogramm finden Sie beide Varianten.

1.9.5. Den Dialog mit ESC beenden

Ihnen ist sicher bekannt, dass sich die meisten Dialogboxen mit der ESC-Taste einfach schließen lassen. Diese Funktionalität fehlt bisher noch, lässt sich aber recht einfach nachrüsten. Wenn Sie auf ESC drücken, sendet das System die Nachricht "`WM_COMMAND`". Der `wParam` enthält dann den Wert `IDCANCEL`, so dass Sie nur wie folgt reagieren müssen:

```
WM_COMMAND:
  if(wParam = IDCANCEL) then SendMessage(hDlg,WM_CLOSE,0,0)
    else { ... }
```

Mehr ist nicht erforderlich.

1.9.6. Auf Dialogelemente zugreifen

Für die Arbeit mit den Dialog-Elementen bietet das API ein paar spezielle Funktionen. Sie dürfen nicht vergessen, dass z.B. eine Checkbox nun kein eigenständig erzeugtes Fenster sondern ein Teil der Dialog-Ressource ist. Sie müssen also zunächst das Handle des jeweiligen Elementes herausfinden; erst dann können Sie es ansprechen.

Eine der wichtigsten Funktionen dafür ist "`GetDlgItem`" Bleiben wir beim Beispiel der Checkbox: Sagen wir, wir haben ihr die ID `IDC_CHECKBOX1` gegeben, dann würden wir mit folgender Zeile ihr Handle bekommen:

```
hDlgCheckBox := GetDlgItem(hDlg,IDC_CHECKBOX1);
```

Und ab hier unterscheidet sich der Zugriff auf die Eigenschaften des jeweiligen Controls in nichts mehr von dem, was wir bereits in den jeweiligen Tutorials besprochen haben.

GetDlgItem-Definition

```
HWND GetDlgItem(
  HWND hDlg,        // Dialog-Handle
  int nIDDlgItem    // ID des gesuchten Elements
);
```

Eine andere Funktion ist "`SendDlgItemMessage`". Sie funktioniert genau wie "`SendMessage`", nur dass Sie hier direkt ein Dialog-Element ansprechen können. Wenn wir also den Haken in unsere Checkbox setzen wollen, dann sieht das so aus:

```
SendDlgItemMessage(hDlg,IDC_CHECKBOX1,BM_SETCHECK,BST_CHECKED,0);
```

Wir geben also zusätzlich die ID des Steuerelementes an.

SendDlgItemMessage-Definition

```
LONG SendDlgItemMessage(
  HWND hDlg,        // Dialog-Handle
  int nIDDlgItem,   // ID des gesuchten Elements
  UINT Msg,         // Message, die wir senden wollen
  WPARAM wParam,    // erster Message-Parameter
  LPARAM lParam     // zweiter Message-Parameter
);
```

Zum Abschluss wollen wir uns ansehen, wie man Text in ein Dialog-Element schreibt und wieder ausliest. Dazu bietet das API die Funktionen "`SetDlgItemText`" und "`GetDlgItemText`".

Wenn wir einen Text schreiben wollen, dann benötigen wir eigentlich nur das Dialog-Handle und die ID des gewünschten Controls - und den Text, natürlich!

```
SetDlgItemText(hDlg, IDC_HELLO, 'Hello World!');
```

Zum Auslesen brauchen wir `pchar`-Variable, die den Text aufnimmt. Dialog-Handle und Control-ID sind auch hier wieder erforderlich:

```
ZeroMemory(@szBuffer,sizeof(szBuffer));
GetDlgItemText(hDlg,IDC_HELLO,szBuffer,sizeof(szBuffer));
MessageBox(0,szBuffer,'Ergebnis',MB_OK or MB_ICONINFORMATION);
```

SetDlgItemText-Definition

```
BOOL SetDlgItemText(
  HWND hDlg,        // Dialog-Handle
  int nIDDlgItem,   // ID des gesuchten Elements
  LPCTSTR lpString // Text
);
```

GetDlgItemText-Definition

```
UINT GetDlgItemText(
  HWND hDlg,        // Dialog-Handle
  int nIDDlgItem,   // ID des gesuchten Elements
  LPTSTR lpString, // Zeiger auf einen Textpuffer
  int nMaxCount     // Größe des Puffers
);
```

1.10. Problem mit TAB und Alt+<Shortcut>

Wenn Sie die Arbeit mit der VCL gewohnt sind, dann werden Sie sicher wissen, dass Sie mit der TAB-Taste von einem Control zum nächsten wechseln können. Und dass Sie die unterstrichenen Buchstaben eines Controls in Verbindung mit der ALT-Taste als Shortcut benutzen können, um die entsprechende Funktion zu aktivieren.
Ohne VCL sieht die Sache anders aus. Wenn Sie den Beispielen gefolgt sind und diese getestet haben, dann werden Sie merken, dass genau diese Funktionalität fehlt.

Um sie nachzurüsten, müssen wir die Funktion "IsDialogMessage" aus dem Dialog-Tutorial ein wenig zweckentfremden. Nun ist diese Funktion zwar für Dialogfelder gedacht, aber in unserem Fall ist sie hilfreich. Da sie das Handle eines Fensters erwartet, müssen wir unser, mit "`CreateWindowEx`" erzeugtes Hauptfenster angeben:

```
{Fenster erzeugen}
hwndMain := CreateWindowEx(0, ClassName, AppName, WS_CAPTION or WS_VISIBLE or
WS_SYSMENU,
  integer(CW_USEDEFAULT),integer(CW_USEDEFAULT),WindowWidth,
  WindowHeight, 0, 0, hInstance, nil);

while GetMessage(msg,0,0,0) do
  begin
    if not(IsDialogMessage({Fensterhandle ->} hwndMain, msg)) then
      begin
        TranslateMessage(msg);
        DispatchMessage(msg);
      end;
  end;
```

Wenn es sich bei unserer ankommenden Nachricht also um keine Dialognachricht handelt, dann wird sie auf dem normalen Weg weitergereicht und bearbeitet, ansonsten springt der Dialogmanager ein und sorgt dafür, dass unser Programm nun auch auf die Tabulatortaste reagiert.

Dazu ist allerdings noch ein Schritt erforderlich: Sie müssen allen Controls, die Sie auf diesem Weg ansteuern wollen, das Stilattribut `WS_TABSTOP` zuordnen. Beispielsweise:

```
hClose := CreateWindowEx(WS_EX_CLIENTEDGE, 'BUTTON', '&Schließen',
  WS_VISIBLE or WS_CHILD { neues Stilattribut ->} or WS_TABSTOP,
  WindowWidth-115, WindowHeight-60, 100, 25,
  hWnd, IDC_CLOSE, hInstance, nil);

CreateWindowEx(WS_EX_CLIENTEDGE, 'BUTTON', 'Button 1',
  WS_VISIBLE or WS_CHILD { neues Stilattribut ->} or WS_TABSTOP, 10, 10, 100,
25,
  hWnd, 0, hInstance, nil);
```

Wenn Sie das Programm jetzt testen, können Sie die einzelnen Buttons nacheinander mit der TAB-Taste anwählen. Und durch den Dialogmanager funktioniert auch das Schließen über den Button-Shortcut ALT+S.

Hinweis

Ich bitte zu bedenken, dass dies keine offizielle Lösung ist. Es ist mehr ein Patch, wenn Sie die TAB-Funktionalität unbedingt in Ihren selbst erstellten Fenstern benötigen. Und ich habe feststellen müssen, dass es bei Menüs zu Problemen kommen kann, wenn man versucht, mit ALT und dem unterstrichenen Buchstaben das Menü zu öffnen. Manchmal löst auch der Druck von ESC eine willkürlich (?) gewählte Aktion aus.

Wenn möglich, benutzen Sie bitte Dialog-Ressourcen anstelle von Fenstern. Dann können Sie den Dialogmanager des Systems nutzen und müssen sich keine Gedanken über die Implementierung der TAB/ALT-Funktionalität machen.

2. Standarddialoge

Dieses Kapitel befasst sich mit den Standarddialogen des Systems. Dazu gehören z.B. der Öffnen- und der Speichern-Dialog, der Schriftartendialog, usw. Das System stellt Ihnen die Funktionalität zur Verfügung, und Sie müssen sich weder um Dialogressourcen noch um die Nachrichtenverarbeitung kümmern.

Nutzen wir also die Gelegenheit und entwickeln im Zuge dieses Kapitels einen kleinen Editor, der uns die wichtigsten Dialoge vorstellen wird. Nicht mehr eingehen werde ich auf das Erzeugen des Fensters des Hauptmenüs. Das Editieren des Textes wird durch ein Eingabefeld möglich, wobei lediglich ein paar neue Stilattribute für mehrzeiligen Text und die evtl. erforderlichen Scrollbalken zu beachten sind.

2.1. Dateien öffnen und speichern

Was wäre ein Editor wohl ohne die Möglichkeiten, vorhandene Dateien zu öffnen und die bearbeiteten oder neuen Daten wieder zu speichern?

Neben den beiden Dialogen wird in diesem Tutorial ein Hauptaugenmerk auf den Funktionen "`ReadFile`" und "`WriteFile`" liegen, die wir zum Laden und Speichern der Textdateien unseres Mini-Editors nutzen werden.

2.1.1. Das TOpenFileName-Record

Das Record `TOpenFileName` wird sowohl vom Öffnen- als auch vom Speicherndialog verwendet. Es bietet uns (über Flags u.ä.) Zugriff auf die Gestaltung des jeweiligen Dialoges, und natürlich brauchen wir sie um den Dateinamen in Erfahrung zu bringen.
Es ist mit den Eigenschaften der VCL-Komponenten **TOpenDialog** und **TSaveDialog** vergleichbar, und wenn Sie bereits Erfahrungen mit diesen Komponenten haben, werden Ihnen viele Variablen von `TOpenFileName` bekannt vorkommen.

Wie bei einigen anderen Systemfunktionen müssen Sie das Record erst initialisieren, indem Sie ihm seine eigene Größe zuweisen:

```
ofn.lStructSize := SizeOf(TOpenFileName);
```

Wenn Sie die Dialoge bereits mit der VCL verwendet haben, dann sind Sie daran gewöhnt, dass die einzelnen Elemente des Dateifilters durch die Pipe (den senkrechten Strich) voneinander getrennt sind. Beim Aufruf der API-Funktion verwenden Sie stattdessen das Zeichen Null als Trenner, und Sie schließen die Angabe mit zwei Null-Zeichen ab:

```
const
  Filter = 'Textdateien (*.txt)'#0'*.txt'#0'Alle Dateien (*.*)'#0'*.*'#0#0;

{ ... }

ofn.lpstrFilter := Filter;
```

Benutzt ein Typ mehrere Dateiendungen, müssen diese durch ein Semikolon voneinander getrennt sein, beispielsweise:

```
const
  Filter = 'Pascal-Dateien (*.pas, *.dpr, *.inc)'#0'*.txt;*.dpr;*.inc'#0#0;
```

Und sofern wir nichts Besonders vorhaben, genügen die folgenden Flags vollkommen:

```
ofn.Flags       := OFN_FILEMUSTEXIST or OFN_PATHMUSTEXIST or OFN_LONGNAMES;
```

Bitte ziehen Sie auch die Hilfe oder das MSDN für die weiteren Möglichkeiten zu Rate. Und wenn Sie z.B. ein VCL-Programm auf API-Basis umschreiben wollen, dann ist es durchaus hilfreich, sich die Parameter der Dialog-Komponente anzusehen und mit den Erklärungen der Hilfe zu vergleichen.

TOpenFileName-Definition

```
typedef struct tagOFN {
  DWORD         lStructSize;        // Größe des Records
  HWND          hwndOwner;          // Handle des übergeordneten Fensters
  HINSTANCE     hInstance;          // Programminstanz
  LPCTSTR       lpstrFilter;        // VCL = "Filter"
  LPTSTR        lpstrCustomFilter;
  DWORD         nMaxCustFilter;
  DWORD         nFilterIndex;       // VCL = "FilterIndex"
  LPTSTR        lpstrFile;          // VCL = "FileName"
  DWORD         nMaxFile;           // Größe des Puffers für den Dateinamen
  LPTSTR        lpstrFileTitle;
  DWORD         nMaxFileTitle;
  LPCTSTR       lpstrInitialDir;    // VCL = "InitialDir"
  LPCTSTR       lpstrTitle;         // VCL = "Title"
  DWORD         Flags;              // VCL = "Options"
  WORD          nFileOffset;        // Beginn des Dateinamens im Pfad
  WORD          nFileExtension;     // Beginn der Dateierweiterung im Pfad
  LPCTSTR       lpstrDefExt;        // VCL = "DefaultExt"
  LPARAM        lCustData;
  LPOFNHOOKPROC lpfnHook;
  LPCTSTR       lpTemplateName;
#if (_WIN32_WINNT >= 0x0500)
  void *        pvReserved;
  DWORD         dwReserved;
  DWORD         FlagsEx;
#endif // (_WIN32_WINNT >= 0x0500)
} OPENFILENAME
```

Hinweis für Windows 2000 und XP:

Die Syntaxdefinition zeigt bereits das erweiterte Record, das für Windows 2000 und XP genutzt werden kann. Wenn Ihre Delphi-Version aktuell genug ist, steht Ihnen dieses neue Record bereits zur Verfügung. Andernfalls ergänzen Sie bitte am Ende Ihres Records die drei neuen Membervariablen:

```
type
  tagOFNA = packed record
    { alte Recordvariablen }

    pvReserved: pointer;
    dwReserved: dword;
    FlagsEx: dword;
  end;
```

Bei der Unicodeversion (`tagOFNW`) ist genau so zu verfahren. - Die beiden Membervariable `pvReserved` und `dwReserved` sind bisher allerdings reserviert und müssen auf **nil** bzw. Null gesetzt werden. Lediglich die `FlagsEx`-Variable steht Ihnen zur Verfügung. Benutzen Sie hier das Flag `OFN_EX_NOPLACESBAR` zum Ausblenden der so genannten "Places bar" im Dialog:

```
ofn.FlagsEx := OFN_EX_NOPLACESBAR;
```

Wenn Sie mit dem neuen Record arbeiten, dann müssen Sie unter Windows 95, 98, ME und NT die alte Recordgröße initialisieren, da diese Systeme nichts mit den neuen Variablen anfangen können. Dazu dienen die folgenden Konstanten:

```
const
  {$EXTERNALSYM OPENFILENAME_SIZE_VERSION_400A}
  OPENFILENAME_SIZE_VERSION_400A = sizeof(TOpenFileNameA) -
    sizeof(pointer) - (2 * sizeof(dword));
  {$EXTERNALSYM OPENFILENAME_SIZE_VERSION_400W}
  OPENFILENAME_SIZE_VERSION_400W = sizeof(TOpenFileNameW) -
    sizeof(pointer) - (2 * sizeof(dword));
  {$EXTERNALSYM OPENFILENAME_SIZE_VERSION_400}
  OPENFILENAME_SIZE_VERSION_400  = OPENFILENAME_SIZE_VERSION_400A;
```

Der Aufruf könnte sich, betriebssystemabhängig, dann wie folgt gestalten:

```
if(Win2k) or (WinXP) then ofn.lStructSize := sizeof(TOpenFileName)
  else ofn.lStructSize := OPENFILENAME_SIZE_VERSION_400;
```

Hinweis

Diesem Tutorial liegt eine spezielle Unit bei ("`CommDlg_Fragment.pas`"), die das erweiterte Record und die neuen Konstanten enthält. Benutzen Sie diese Unit, wenn Ihre Delphi-Version die neuen Variablen nicht kennt.

2.1.2. Den Öffnen- und Speicherndialog aufrufen

Nachdem das Record nun gefüllt ist, können wir nun den gewünschten Dialog aufrufen. Dazu dienen die Funktionen "`GetOpenFilename`"

```
if GetOpenFileName(ofn) then
  MessageBox(0,buffer,'gewählte Datei',MB_OK or MB_ICONINFORMATION);
```

und "`GetSaveFilename`":

```
if GetSaveFileName(ofn) then
  MessageBox(0,buffer,'angegebener Dateiname:',MB_OK or MB_ICONINFORMATION);
```

Jede dieser Funktionen liefert entweder **true** (wenn der Anwender mit OK bestätigt) oder **false** zurück.

2.1.3. Die Datei öffnen und einlesen

Bleiben wir bei dem Beispiel, dass wir eine Datei mit dem Öffnen-Dialog ausgewählt haben und nun im Eingabefeld unseres Mini-Editors anzeigen möchten. Dazu holen wir uns zunächst ein Handle auf die Datei und fordern Speicher an:

```
hFile := CreateFile(buffer, GENERIC_READ  or GENERIC_WRITE,
  FILE_SHARE_READ or FILE_SHARE_WRITE, nil, OPEN_EXISTING,
  FILE_ATTRIBUTE_NORMAL, 0);

{ Speicher anfordern }
hMemory := GlobalAlloc(GMEM_MOVEABLE or GMEM_ZEROINIT, MEMSIZE);
```

Wenn das erledigt ist, holen wir uns die Adresse des ersten Bytes des eben angelegten Speicherobjektes

```
pMemory := GlobalLock(hMemory);
```

lesen die Datei und übertragen Sie mit "`WM_SETTEXT`" in unser Eingabefeld:

```
ReadFile(hFile, pMemory^, MemSize-1, SizeReadWrite, nil);

SendMessage(Edit, WM_SETTEXT, 0, Integer(pMemory));
SendMessage(Edit, EM_SETSEL, 0, 0);
```

Und zum Schluss wird - wie in jedem guten Haushalt - aufgeräumt:

```
CloseHandle(hFile);
GlobalUnlock(CARDINAL(pMemory));
GlobalFree(hMemory);
```

Zum Lesen der Daten (in dem Fall max. 64k) benutzen wir die Funktion "`ReadFile`". Von besonderer Bedeutung ist `lpNumberOfBytesRead` (im Code: "`SizeReadWrite`"). Er liefert uns die Anzahl der tatsächlich gelesenen Bytes zurück. Zwar wird dieser Wert im Beispiel nicht weiter beachtet, aber er kann in Ihren Programmen der Fehlerkontrolle dienen.

ReadFile-Definition

```
BOOL ReadFile(
  HANDLE hFile,                // Dateihandle
  LPVOID lpBuffer,             // Zeiger auf den Datenpuffer
  DWORD nNumberOfBytesToRead,  // Anzahl der zu lesenden Bytes
  LPDWORD lpNumberOfBytesRead, // Anzahl der gelesenen Bytes
  LPOVERLAPPED lpOverlapped
);
```

2.1.4. Den Inhalt des Eingabefeldes als Datei speichern

Wenn wir den Text des Editors speichern wollen, dann machen wir im Prinzip alles rückwärts. Zuerst öffnen wir die Datei natürlich wieder, fordern Speicher an und holen uns die Adresse zu unserem Speicher. So weit, so gut.
Nun können wir gleich zwei Fliegen mit einer Klappe schlagen: die Nachricht "`WM_GETTEXT`" liefert uns als Rückgabewert die Anzahl der gelesenen Bytes (= Zeichen), und sie schreibt uns die Daten auch gleich noch in den Puffer:

```
{ zu schreibende Bytes ermitteln }
SizeReadWrite := SendMessage(Edit, WM_GETTEXT, MEMSIZE-1, Integer(pMemory));

{ Datei schreiben }
WriteFile(hFile, pMemory^, SizeReadWrite, SizeReadWrite, nil);
```

Selbstverständlich müssen wir auch hier wieder aufräumen, d.h. die Datei schließen und den benutzten Speicher freigeben.

WriteFile-Definition

```
BOOL WriteFile(
  HANDLE hFile,                    // Dateihandle
  LPCVOID lpBuffer,                // Zeiger auf den Datenpuffer
  DWORD nNumberOfBytesToWrite,     // Anzahl der zu schreibenden Bytes
  LPDWORD lpNumberOfBytesWritten,  // Anzahl der geschriebenen Bytes
  LPOVERLAPPED lpOverlapped
);
```

2.1.5. Mehr als eine Datei auswählen

Da es auch Anwendungen gibt, bei denen man mehr als eine Datei laden kann, soll dies nun kurz demonstriert werden. Folgendes ist dabei zu beachten: wir müssen die `Flags`-Membervariable um ein Flag erweitern, was aber zur Folge hat, dass der Dialog dann im "alten" Look von Windows 3.x angezeigt wird:

```
ofn.Flags := { ... } or OFN_ALLOWMULTISELECT;
```

Dies können Sie vermeiden, indem Sie zusätzlich das Flag `OFN_EXPLORER` benutzen und so den von Windows 9x/NT/ME/2000/XP gewohnten Stil erhalten:

```
ofn.Flags := { ... } or OFN_ALLOWMULTISELECT or OFN_EXPLORER;
```

Weil es aber in beiden Varianten Unterschiede im Rückgabeergebnis gibt, müssen wir uns auch beides ansehen.

2.1.5.1. Der alte Dialogstil

In diesem Fall zeigt der Textpuffer auf einen String, der sowohl den Namen des Ordners als auch die Namen der ausgewählten Dateien enthält, wobei alles durch ein Leerzeichen voneinander getrennt ist. Das könnte also z.B. so aussehen:

```
C:\Eigene~1 Datei1.txt Datei2.txt Datei3.txt
```

Sie müssen den String daher entsprechend parsen, wenn Sie den Dateinamen und den Ordnernamen in Ihrer Anwendung benutzen wollen.

2.1.5.2. Der neue Dialogstil

Ein wenig anders sieht das beim Win9x-Stil der Dialogbox auf. Auch hier enthält der Textpuffer nur einen String. Doch diesmal sind die einzelnen Angaben durch das Zeichen `#0` voneinander getrennt. Das hat ursächlich mit den langen Dateinamen zu tun. Im Vergleich mit dem o.g. String der alten Dialogbox würde der Ordnername diesmal als `C:\Eigene Dateien` zurückgeliefert werden. Der letzte Dateiname endet mit zwei `#0`-Zeichen.
Durch die Trennung mit den `#0`-Zeichen können Sie den Puffer daher nicht direkt weiterverarbeiten; Sie würden lediglich den Namen des Ordners sehen:

```
MessageBox(0,buffer,nil,0);
```

An die Dateinamen kommen Sie aber mit Hilfe einer `pchar`-Variablen heran, die Sie auf den Beginn des Puffers setzen:

```
p := buffer;
```

Nun gehen Sie solange durch den Textpuffer, bis das erste Zeichen dieser Variable `#0` entspricht. Dazu eignet sich am besten eine **while**-Schleife:

```
while(p[0] <> #0) do begin
  MessageBox(Handle,p,nil,0);
```

Auch dieser Aufruf würde zuerst den Ordnernamen anzeigen. Mit Hilfe von `inc` erhöhen wir jedoch die Position der `pchar`-Variablen im Puffer um die Länge des aktuellen Strings (+1, da er ja mit einem `#0`-Zeichen endet):

```
  inc(p,lstrlen(p)+1);
end;
```

Auf diese Weise erhalten wir nacheinander die einzelnen Dateinamen und können sie mit dem Namen des Ordners benutzen.

2.2. Die Schriftart ändern

Unser Editor benutzt bisher beim Start die Systemschrift zur Anzeige:

```
hWndFont := GetStockObject(SYSTEM_FONT);
if(hWndFont <> 0) then
  SendMessage(hEdit, WM_SETFONT, hWndFont, 1);
```

Da diese Schrift nun vielleicht nicht jedem gefallen wird, und da es mittlerweile bei Editoren üblich ist, dass man die Schriftart selbst wählen kann, soll auch unser Editor um diese Funktionalität erweitert werden.

2.2.1. Das TChooseFont-Record

Für den Schriftartendialog benötigen wir ein Record namens `TChooseFont`, das uns Zugriff auf den Dialog bietet. Zuerst müssen wir es initialisieren und ihm seine eigene Größe und das Parent-Fenster zuweisen:

```
cf.lStructSize := sizeof(TCHOOSEFONT);
cf.hWndOwner   := wnd;
```

Des Weiteren benötigen wir eine Variable vom Typ `TLogFont` (im Beispiel "`lf`"), die wir an die entsprechende Membervariable des Records weitergeben:

```
cf.lpLogFont   := @lf;
```

Sofern die Variable "`lf`" nun eine gültige Schriftart enthält, wird diese im Dialog bereits gesetzt. Und beim Beenden mit OK wird die Variable entsprechend unserer Auswahl angepasst, so dass wir die Schriftart in unserem Editor recht einfach ändern können. In diesem Fall müssen wir aber das Flag "`CF_INITTOLOGFONTSTRUCT`" setzen, weil sonst die `lpLogFont`-Membervariable nicht berücksichtigt wird:

```
cf.Flags       := CF_SCREENFONTS or CF_INITTOLOGFONTSTRUCT or
  CF_NOSCRIPTSEL;
```

Weitere Flags finden Sie im MSDN und/oder im PSDK. In unserem Fall zeigt der Dialog lediglich die Schriftarten, -stile und -größen an.

TChooseFont-Definition

```
typedef struct {
    DWORD lStructSize;      // Größe des Records
    HWND hwndOwner;         // Handle des Parent-Fensters
    HDC hDC;
    LPLOGFONT lpLogFont;    // Zeiger auf "TLogFont"-Variable
    INT iPointSize;
    DWORD Flags;            // Flags
    COLORREF rgbColors;
    LPARAM lCustData;
    LPCFHOOKPROC lpfnHook;
    LPCTSTR lpTemplateName;
    HINSTANCE hInstance;
    LPTSTR lpszStyle;
    WORD nFontType;
    INT nSizeMin;
    INT nSizeMax;
} CHOOSEFONT
```

2.2.2. Den Schriftartendialog aufrufen

Nachdem wir nun das `TChooseFont`-Record gefüllt haben, können wir den Dialog aufrufen, dem wir das Record als Parameter übergeben:

```
if(ChooseFont(cf)) then
  Result := CreateFontIndirect(lf);
```

Das Ergebnis ist **true**, wenn der Anwender den Dialog mit OK bestätigt hat. In diesem Fall wird die von uns zuvor zugewiesene `TLogFont`-Variable entsprechend der Auswahl angepasst, so dass wir mit Hilfe der Funktion "`CreateFontIndirect`" einen logischen Font erzeugen können. Rückgabewert der Funktion ist ein Handle auf den Font.

Im Beispielprogramm wird dieser logische Font dann benutzt, um die Schriftart im Editfeld zu ändern:

```
myfont := SelectFont(wnd);
if(myfont <> 0) then
  begin
    SendMessage(hEdit,WM_SETFONT,WPARAM(myfont),LPARAM(false));
    UpdateWindow(hEdit);
  end;
```

2.3. Suchen- / Ersetzendialog

2.3.1. Vorarbeit

Bevor wir den Suchen- und den Ersetzendialog nutzen können, müssen wir (leider) ein paar Änderungen am Programm vornehmen. Die Textsuche wird nämlich nur von RichEdit-Controls unterstützt, daher müssen wir unser Programm damit ausrüsten. Aber der Aufwand hält sich in Grenzen:

```
program editor;

uses
  { ... },
  RichEdit;

begin
    LoadLibrary('riched20.dll'); // RichEdit 2.0
    LoadLibrary('riched32.dll'); // RichEdit

  { ... }
end.
```

Die Rückgabeergebnisse von "`LoadLibrary`" interessieren uns nicht, da wir im Fehlerfall immer noch das normale Edit-Control erzeugen werden. Folgendes ist dabei zu sagen: Das RichEdit 2.0 unterstützt beim Suchen die Auswahl der Suchrichtung. Wir können also vom Textanfang in Richtung Textende suchen, oder umgekehrt.
Deshalb versuchen wir zuerst auch, ein RichEdit 2.0 zu erzeugen. Funktioniert das nicht, erstellen wir ein normales RichEdit-Control. Klappt das auch nicht, bleiben wir bei unserem Multi-Line-Edit-Control:

```
case uMsg of
  WM_CREATE:
    begin
      { ... }

      { RichEdit 2.0 erzeugen }
      hEdit := CreateWindowEx(WS_EX_CLIENTEDGE,'RichEdit20A',nil,WS_CHILD or
        WS_VISIBLE or ES_MULTILINE or ES_NOHIDESEL or WS_VSCROLL or
        ES_AUTOVSCROLL,0,0,0,0,wnd,IDC_EDIT,hInstance,nil);
      RichEdit20 := (hEdit <> 0);

      { RichEdit 1.0 erzeugen, wenn R2.0 nicht möglich ist }
      if(not RichEdit20) then
        hEdit := CreateWindowEx(WS_EX_CLIENTEDGE,'RICHEDIT',nil,WS_CHILD or
          WS_VISIBLE or ES_MULTILINE or ES_NOHIDESEL or WS_VSCROLL or
          ES_AUTOVSCROLL,0,0,0,0,wnd,IDC_EDIT,hInstance,nil);
      RichEdit10 := (hEdit <> 0);

      { es ist auch kein RichEdit 1.0 möglich, :o( }
      if(not RichEdit10) and (not RichEdit20) then
        hEdit := CreateWindowEx(WS_EX_CLIENTEDGE, 'Edit', '', WS_VISIBLE or
          WS_CHILD or ES_MULTILINE or WS_VSCROLL or ES_AUTOVSCROLL, 0, 0, 0, 0,
          wnd, IDC_EDIT, hInstance, nil);
      SetFocus(hEdit);

      { ... }
    end;
end;
```

Und nur wenn min. eins der beiden RichEdit-Controls (v2.0 oder v1.0) erzeugt werden konnte, erweitern wir das Hauptmenü des Editors um die Punkte "Suchen" und "Ersetzen":

```
if(RichEdit10) or (RichEdit20) then begin
  AppendMenu(hSubmenu,MF_STRING,IDM_SEARCH,'Suchen');
  AppendMenu(hSubmenu,MF_STRING,IDM_REPLACE,'Ersetzen');
end;
```

"RichEdit10" und "RichEdit20" sind zwei globale bool-Variablen, die hauptsächlich der Kontrolle dienen.

2.3.2. Den Suchen- oder den Ersetzendialog aufrufen

Zum Aufruf der beiden Dialoge, benötigen wir zuerst ein Record vom Typ TFindReplace. Außerdem brauchen wir noch einen Textpuffer, der den gesuchten String aufnimmt:

```
var
  fnd            : TFindReplace;
  FindStr        : array[0..4096]of char;
  FindLen        : DWORD = sizeof(FindStr) - 1;
```

Die Größe des Records muss, natürlich, wieder festgelegt werden

```
fnd.lStructSize    := sizeof(TFindReplace);
```

Als Besitzerfenster geben wir das Handle unseres Editorfensters an. Gleichzeitig setzen wir ein Flag, das aber vom verwendeten RichEdit abhängig ist. Konnte ein RichEdit 2.0 erzeugt werden, setzen wir die Suchrichtung auf "Abwärts". Ist nur ein RichEdit 1.0 verfügbar, dann setzen wir das Abwärtsflag natürlich auch (um Fehler zu vermeiden), verbergen aber die Auswahl der Richtung:

```
fnd.hWndOwner := wnd;
if(RichEdit20) then fnd.Flags := FR_DOWN
  else fnd.Flags := FR_HIDEUPDOWN or FR_DOWN;
```

Das Programm benutzt zwar eine etwas andere Methode, aber das Prinzip bleibt das selbe.
Zu guter Letzt weisen wir noch den Textpuffer zu

```
fnd.lpstrFindWhat := FindStr;
fnd.wFindWhatLen  := FindLen;
```

Bevor wir den Dialog nun tatsächlich aufrufen können, müssen wir den Wert der so genannten FINDMSGSTRING-Nachricht herausfinden. Diese ist vom System bereits vorgeben, und mit Hilfe der Funktion "RegisterWindowMessage" können wir den numerischen Wert ermitteln:

```
var
  FindTextMsgId : UINT;

{ ... }

FindTextMsgId      := RegisterWindowMessage(FINDMSGSTRING);
```

Diese Nachricht wird später an unser Fenster gesendet. Um dann entsprechend reagieren zu können, benötigen wir den numerischen Wert.
Jetzt können wir den Suchendialog aber erst einmal aufrufen:

```
CommDlg.FindText(fnd);
```

Wollen wir stattdessen den Ersetzendialog aufrufen, benötigen wir zunächst einen weiteren Textpuffer:

```
var
  ReplStr : array[0..4096]of char;
```

Dass sich der Aufruf nicht groß vom Suchendialog unterscheidet, erkennen Sie bereits daran, dass unser Editor in beiden Fällen die Prozedur "`Find`" benutzt. Wir müssen lediglich den neuen Textpuffer zuweisen:

```
if(fReplaceMode) then begin
  fnd.lpstrReplaceWith := ReplStr;
  fnd.wReplaceWithLen  := FindLen;
end;
```

und die Flags anpassen, da wir den Text grundsätzlich in Richtung Textende suchen und ersetzen wollen:

```
if(RichEdit20) and (not fReplaceMode) then fnd.Flags := FR_DOWN
  else fnd.Flags := FR_DOWN or FR_HIDEUPDOWN;
```

Und bis auf den Aufruf des Dialogs bleibt alles andere unverändert:

```
CommDlg.ReplaceText(fnd);
```

TFindReplace-Definition

```
typedef struct {
    DWORD lStructSize;       // Record-Größe
    HWND hwndOwner;          // Besitzerfenster
    HINSTANCE hInstance;
    DWORD Flags;             // Flags
    LPTSTR lpstrFindWhat;    // Suchtext
    LPTSTR lpstrReplaceWith; // neuer Text (nur für Ersetzen)
    WORD wFindWhatLen;       // Länge des Suchtextes
    WORD wReplaceWithLen;    // Länge des neuen Textes
    LPARAM lCustData;
    LPFRHOOKPROC lpfnHook;
    LPCTSTR lpTemplateName;
}
```

2.3.3. Text suchen

Im letzten Kapitel wurde es bereits erwähnt: wenn wir den Suchendialog ausführen (in dem Sinn, dass wir Text eingeben und auf den Button "Weitersuchen" klicken), dann wird an unser Fenster die Nachricht `FINDMSGSTRING` gesendet. Dies ist keine Konstante, sondern wir mussten den Wert dieser Nachricht zuerst ermitteln.
Wir sollten daher vor der Bearbeitung der Nachricht prüfen, ob überhaupt ein Wert zugewiesen wurde. Der `lParam`-Wert ist dann ein Zeiger auf ein `TFindReplace`-Record:

```
case uMsg of
  { ... }

  // Suchfunktion, by NicoDE
  else if(FindTextMsgId <> 0) and (uMsg = FindTextMsgId) then begin
    FindParam := PFindReplace(lp);
```

Die `Flags`-Membervariable dieses Records enthält nun den Wert `FR_FINDNEXT`, wenn wir im Suchendialog den Button "Weitersuchen" anklicken. In diesem Fall rufen wir die eigentliche Suchfunktion, "`SearchText`" auf, der wir neben Besitzerfenster und Suchtext auch die Flags übergeben, mit denen wir im Suchendialog festlegen können, in welcher Richtung wir suchen wollen, ob ganze Wörter gesucht werden sollen und ob die Groß- und Kleinschreibung eine Rolle spielt:

```
    if(FindParam.Flags and FR_FINDNEXT = FR_FINDNEXT) then
      SearchText(FindParam.hWndOwner,FindParam.lpstrFindWhat,
        FindParam.Flags and FR_DOWN = FR_DOWN,
        FindParam.Flags and FR_MATCHCASE = FR_MATCHCASE,
        FindParam.Flags and FR_WHOLEWORD = FR_WHOLEWORD);
  end
```

Da wir stets mit dem Besitzerfenster arbeiten, unserem Editorfenster also, müssen wir in der Suchfunktion zuerst das Handle des RichEdit-Controls ermitteln. Dazu dient uns die von den Dialogen bekannte Funktion "`GetDlgItem`":

```
rEdit := GetDlgItem(wnd,IDC_EDIT);
```

Dann stellen wir fest, ob in dem Control bereits Text markiert ist. Dabei hilft uns die Nachricht "`EM_EXGETSEL`", die als `lParam` einen Zeiger auf ein `CHARRANGE`-Record erwartet. Da wir den Text noch suchen wollen, benutzen wir hier gleich die entsprechende Membervariable eines `TFindText`-Records:

```
SendMessage(rEdit,EM_EXGETSEL,0,LPARAM(@FindRec.chrg));
```

Die `cpMax`-Membervariable setzen wir vor jedem Aufruf auf -1, bei der `cpMin`-Variable kommt es auf unsere Suchrichtung an. Suchen wir in Richtung Textende (Abwärts), dann genügt es, den alten Wert von `cpMax` einstellen. Suchen wir in Richtung Textanfang (Aufwärts), dann sollten wir den alten Wert von `cpMin` beibehalten.

```
if(Down) then FindRec.chrg.cpMin := FindRec.chrg.cpMax;
FindRec.chrg.cpMax := -1;
```

Warum?
Stellen wir uns dazu einfach einen kleinen Beispieltext vor, den wir nach irgendeinem Wort durchsuchen; sagen wir: *Tutorial*. Wir werden gleich noch sehen, wie der gefundene Text markiert wird. Und genau das ist der Punkt! Die o.g. Nachricht liefert nun die Markierung zurück, wobei `cpMin` natürlich den Beginn und `cpMax` das Ende des Wortes darstellen - im Fall von *Tutorial* also die Positionen des "T" und "l".
Wollen wir abwärts (in Richtung Textende) suchen, müssen wir als neue Startposition nur den alten Endpunkt einstellen. Auf die Weise geht die Suche beim nächsten Mal hinter dem bereits gefundenen Wort weiter.
Anders sieht es dagegen mit der Suche in Richtung Textanfang aus. Würden wir auch hier den alten Endwert als neuen Startwert einstellen, würde die Suche scheinbar nicht funktionieren. Tatsächlich findet sie jedoch immer das selbe Wort, da wir ja ständig beim Ende des Wortes neu suchen lassen. Behalten wir den Wert von `cpMin` allerdings bei, liefert die Suche auch in der Aufwärtsrichtung mehr als nur ein Ergebnis (sofern der Suchbegriff entsprechend mehrfach vorkommt, natürlich!).

Um Text zu finden, müssen wir diesen natürlich auch angeben; in dem Fall weisen wir ihn dem `TFindText`-Record zu.

```
FindRec.lpstrText     := Text;
```

Von Interesse sind natürlich noch die Flags, die immerhin Auswirkungen auf die Suchergebnisse haben. In welche Richtung suchen wir? Werden nur ganze Wörter gewünscht? Spielt die Schreibweise eine Rolle? All diese Fragen können wir mit Hilfe der Variablen klären, die der Prozedur übergeben wurden:

```
Flags                 := 0;
if(Down) then Flags   := FR_DOWN;
if(Sense) then Flags  := Flags or FR_MATCHCASE;
if(Whole) then Flags  := Flags or FR_WHOLEWORD;
```

Nun, dann suchen wir mal. Das RichEdit bietet zu dem Zweck die Nachricht "`EM_FINDTEXT`", die als `wParam` die eben gesetzten Flags und als `lParam` einen Zeiger auf das zuerst verwendete `TFindText`-Record erwartet. Rückgabewert ist die Position, an der der gesuchte Text gefunden wurde:

```
FindPos := SendMessage(rEdit,EM_FINDTEXT,Flags,LPARAM(@FindRec));
```

Ist der Wert größer Null, dann können wir den gefundenen Text mit Hilfe der Nachricht "`EM_EXSETSEL`" markieren:

```
if(FindPos > 0) then begin
  FindRec.chrg.cpMin := FindPos;
  FindRec.chrg.cpMax := FindPos + lstrLen(Text);
  SendMessage(rEdit,EM_EXSETSEL,0,LPARAM(@FindRec.chrg));
end else begin
  lstrcpy(ErrMsg,pchar('"' + Text + '"'));
  lstrcat(ErrMsg,' kann nicht gefunden werden.');
  MessageBox(wnd,ErrMsg,'editor',MB_ICONINFORMATION);
end;
```

Kann das gesuchte Wort nicht (mehr) gefunden werden, erscheint eine entsprechende Meldung.

CHARRANGE-Definition

```
typedef struct _charrange {
    LONG cpMin;          // Startposition
    LONG cpMax;          // Endposition
}
```

TFindText-Definition

```
typedef struct _findtext {
    CHARRANGE chrg;      // CHARRANGE-Record
    LPCTSTR lpstrText;   // Suchtext
}
```

2.3.4. Text ersetzen

Um Text ersetzen zu können, müssen wir diesen auch erst einmal suchen. Am Prinzip, das wir im letzten Kapitel besprochen haben, ändert sich also nicht viel. Interessant wird es erst, wenn wir den gesuchten alten Text gefunden haben. Wie gehabt, auch er wird markiert:

```
FindRec.chrg.cpMin := FindPos;
FindRec.chrg.cpMax := FindPos + lstrlen(OldText);
SendMessage(rEdit,EM_EXSETSEL,0,LPARAM(@FindRec.chrg));
```

Aber dann ersetzen wir ihn mit Hilfe der Nachricht "`EM_REPLACESEL`" durch den neuen Text:

```
SendMessage(rEdit,EM_REPLACESEL,WPARAM(true),LPARAM(NewText));
```

Der `wParam`-Wert **TRUE** sagt in diesem Fall aus, dass dieses Ersetzen ggf. rückgängig gemacht werden kann. Dazu muss Ihr Programm über eine entsprechende Undo-Funktion verfügen. Der `lParam`-Wert ist, wie Sie sehen können, ein Zeiger auf den Textpuffer mit dem neuen String.

Ebenfalls interessant ist die Gestaltung der Prozedur "`ReplText`". Da der Anwender im Ersetzendialog auch den Button "Alles ersetzen" anklicken kann, müssen wir die Suche in diesem Fall entsprechend oft wiederholen. Ich habe mich für eine **repeat**-Schleife entschieden, die erst verlassen wird, wenn das gesuchte Wort nicht (mehr) vorhanden ist, bzw. wenn der Anwender nur den Button "Ersetzen" benutzt hat:

```
repeat
  { Text suchen }

  if(FindPos > 0) then begin
    { Text ersetzen }
  end else begin
    { Fehlermeldung }
  end;
until(FindPos <= 0) or (not ReplaceAll);
```

Alles, was ich eben erwähnte (das Suchen, das Markieren und das Ersetzen), spielt sich innerhalb dieser Schleife ab.

Ich möchte daher an dieser Stelle noch einmal kurz auf die, auch im letzten Kapitel erwähnte Nachricht `FINDMSGSTRING` eingehen. Natürlich wird sie auch hier gesendet, wenn wir den Ersetzendialog benutzen.
Der Ersetzendialog des Systems bietet übrigens auch einen "Weitersuchen"-Button an. Da wir diese Funktion bereits implementiert haben, müssen wir uns darum nicht kümmern. Es steht allerdings keine Richtungswahl zur Verfügung; es wird also generell in Richtung Textende gesucht.

Um nun die eigentliche Ersetzenfunktion des Editors aufrufen zu können, müssen wir also wieder nur die Flags berücksichtigen. Wir prüfen, ob die Flags `FR_REPLACE` (Button "Ersetzen") oder `FR_REPLACEALL` (Button "Alles ersetzen") vorhanden sind und rufen in dem Fall die neue Prozedur "`ReplText`" auf:

```
if(FindParam.Flags and FR_REPLACE = FR_REPLACE) or
  (FindParam.Flags and FR_REPLACEALL = FR_REPLACEALL) then
ReplText(FindParam.hWndOwner,FindParam.lpstrFindWhat,
  FindParam.lpstrReplaceWith,
  FindParam.Flags and FR_DOWN = FR_DOWN,
  FindParam.Flags and FR_MATCHCASE = FR_MATCHCASE,
  FindParam.Flags and FR_WHOLEWORD = FR_WHOLEWORD,
  FindParam.Flags and FR_REPLACEALL = FR_REPLACEALL);
```

Die unterscheidet sich von der Suchenfunktion aus dem letzten Kapitel eigentlich nur darin, dass hier der neue String, der anstelle des alten eingefügt werden soll, angegeben werden muss, und dass der letzte Parameter darüber entscheidet, ob nur eins oder alle Vorkommen des alten Strings ersetzt werden sollen.

2.3.5. Nacharbeit

Im PSDK von Microsoft findet sich noch das Muss, "`IsDialogMessage`" zu benutzen, wenn man die beiden Dialoge aufruft. Dieser Empfehlung wollen wir im letzten Schritt dieses Kapitels folgen. Dazu benötigen wir zuerst eine globale Variable, die das Handle des Suchendialogs aufnehmen soll:

```
var
  hDlg : HWND = 0;
```

Wenn wir nun im Beispielprogramm die Prozedur "`Find`" aufrufen, weisen wir dort das Handle des benutzten Dialogs der Variablen zu:

```
if(fReplaceMode) then hDlg := CommDlg.ReplaceText(fnd)
  else hDlg := CommDlg.FindText(fnd);
```

Der letzte Schritt ist die Erweiterung der Nachrichtenschleife im Hauptteil des Programms:

```
while(GetMessage(msg,0,0,0)) do
  if(not IsDialogMessage(hDlg,msg)) then begin
    TranslateMessage(msg);
    DispatchMessage(msg);
  end;
```

Nun reagiert der Dialog so, wie wir es aus anderen Anwendungen gewohnt sind. Wir können mit der TAB-Taste zwischen den einzelnen Controls wechseln, mit ENTER die Suche auslösen, ihn mit ESC beenden, usw.

2.4. Der Hilfe-Button

Wenn Sie den Hilfe-Button in den Standarddialogen nutzen wollen, dann müssen Sie zuerst das Flag `*_SHOWHELP` ergänzen. Das Sternchen bedeutet, dass der Präfix vom jeweiligen Dialog abhängig ist (Suchen/Ersetzen = `FR`, Öffnen/Speichern = `OFN`, Schriftart = `CF`).
Außerdem müssen Sie die `hWndOwner`-Variable in jedem Dialog auf das Hauptfenster Ihrer Anwendung setzen. Das ist notwendig, weil das Anklicken des Hilfe-Buttons eine Nachricht auslöst, die von Ihrem Programm bearbeitet werden muss. Als Beispiel ein Auszug aus dem `TOpenFileName`-Record:

```
ofn.hWndOwner := wnd;
```

Wenn Sie sich das Beispielprogramm ansehen, dann werden Sie bemerken, dass der Dialog in einer eigenen Prozedur aufgerufen wird, und das Fensterhandle "`wnd`" wird als Parameter dieser Prozedur übergeben.

Außerdem müssen Sie noch den Wert der so genannten `HELPMSGSTRING`-Nachricht ermitteln. Sie kennen das Prinzip bereits vom Suchen und Ersetzen von Text. Dort haben Sie den Wert einer Nachricht ermittelt, damit Sie Ihr Programm mit der Suchen- und/oder Ersetzenfunktion erweitern konnten.
In der Art läuft das auch hier. Die `HELPMSGSTRING`-Nachricht ist vom System bereits vergeben, und mit Hilfe von "`RegisterWindowMessage`" holen wir uns den numerischen Wert:

```
HelpMsgId := RegisterWindowMessage(HELPMSGSTRING);
```

Wenn Sie nun den Hilfe-Button in einem der Dialoge anklicken, dann wird diese Nachricht an das Fenster gesendet, dass Sie in `hWndOwner` angegeben haben. Da es sich im Fall des Beispiels um das Hauptfenster handelt, können wir die Nachricht entsprechend in der "`WndProc`" des Editorfensters bearbeiten.
Da es sich aber wieder um eine Variable handelt, müssen wir den **else**-Teil der **case**-Anweisung nutzen.

```
case uMsg of
  { ... }

  else if(HelpMsgId <> 0) and (uMsg = HelpMsgId) then
    MessageBox(wnd,pchar(Format('Hier könnte Ihre Hilfe zum %s erscheinen',
      [szHelpMsgArray[iHelpContext]])),nil,0)
  else
    Result := DefWindowProc(wnd, uMsg, wp, lp);
end;
```

Gibt es etwas zu beachten?

Ja: Der `wparam` der Nachricht ist das Handle zum Dialog. Allerdings ist Ihnen bestenfalls das Handle des Suchen- und Ersetzendialoges bekannt, und selbst dabei handelt es sich um zwei verschiedene Dialoge. Eine Abfrage wie

```
if(HWND(wp) = hDlg) then
  { ... }
```

bringt also nicht viel. Zum einen müssten Sie zwei verschiedene Dialog-Variablen benutzen, um zumindest den Suchen- und den Ersetzendialog voneinander unterscheiden zu können. Und zum anderen haben Sie das Handle des Öffnen- und Speichern- und des Schriftartendialogs überhaupt nicht.
Dann ist zu bedenken, dass der `lparam` abhängig vom Dialog ein Zeiger auf das dazu gehörende Record ist. Handelt es

sich also bspw. um den Schriftartendialog, in dem Sie den Hilfe-Button geklickt haben, dann zeigt der `lparam` auf ein `TChooseFont`-Record.

Mit anderen Worten: vergessen Sie die beiden Parameter und nutzen Sie eine einfachere Idee. Der obige Auszug aus dem Beispielprogramm zeigt es Ihnen. Es erscheint eine Messagebox mit einem Text, der den Namen des jeweils angezeigten Dialogs enthält. Das funktioniert mit einem simplen String-Array

```
const
  szHelpMsgArray : array[0..4]of string =
    ('Schriftartendialog',
     'Suchendialog',
     'Suchen- und Ersetzendialog',
     'Öffnendialog',
     'Speicherndialog');
```

auf dessen Elemente mit Hilfe einer `integer`-Variablen zugegriffen wird, die jeweils vor dem Aufruf des Dialogs entsprechend gesetzt wird. Um beim Beispiel mit dem Öffnendialog zu bleiben, hier würde die Zuweisung ganz einfach so aussehen:

```
iHelpContext := 3;
```

Der Rest ist schnell erklärt: Klickt man auf den Hilfe-Button, dann wird die o.g. Nachricht bearbeitet. Und da es sich bei "`iHelpContext`" um eine globale Variable handelt (ebenso wie das String-Array), kann die MessageBox auch den korrekten Dialognamen ergänzen und anzeigen.

Wenn Sie sinnvolle Informationen anzeigen lassen wollen, dann verweise ich Sie hiermit auf den Beitrag Hilfedateien erstellen und nutzen, in dem der Umgang mit Hilfedateien gezeigt wird.

3. CommonControls

Vorwort

Was sind Common Controls?

Alle nachfolgenden Tutorials beziehen sich auf sog. "Common Controls", die in der Bibliothek "`comctl32.dll`" implementiert sind. Sie können diese "allgemeinen Controls" wie nachfolgend beschrieben in Ihren Programmen verwenden. Im Gegensatz zu den Standardcontrols besteht jedoch immer die Gefahr, dass Ihr Betriebssystem die genannte Bibliothek gar nicht oder in einer älteren Version enthält. Möglicherweise sind daher die "Common Controls" auch nicht verfügbar, oder sie weisen einen geringeren Umfang (als hier besprochen) auf.

Des Weiteren sind zwei Schritte erforderlich, damit Sie diese Controls nutzen können:

- Die Unit "`CommCtrl.pas`" muss in der "`USES`"-Klausel stehen.
- Das Programm muss den Befehl "`InitCommonControlsEx`" aufrufen, um die Controls zu initialisieren.

Der "InitCommonControlsEx"-Bug

Was ist der "`InitCommonControlsEx`"-Bug?

Wenn Sie beim Test der beiliegenden Programme bemerken, dass einige der "Common Controls" nicht angezeigt werden, dann liegt dies nicht am Programm. Als Beispiel möchte ich das Eingabefeld für IP-Adressen anführen.

Dazu muss man wissen, dass der Befehl "`InitCommonControls`" im Platform SDK von Microsoft als veraltet (*obsolete*) bezeichnet wird. Die meisten Beispiele dieses Tutorials nutzen ihn dennoch. In einigen Fällen, etwa beim schon erwähnten IP-Control, reicht dieser Befehl aber nicht mehr aus. Hier muss zwingend der Befehl "`InitCommonControlsEx`" verwendet werden. Auf Grund des Bugs werden die Controls jedoch nicht korrekt initialisiert, und das IP-Eingabefeld ist nicht zu sehen.

Welche Versionen sind betroffen?

Delphi-Version	Status
Delphi 2	-
Delphi 3	-
Delphi 4	-
Delphi 5	der Fehler tritt ohne und mit installiertem SP1 auf
Delphi 6	der Fehler tritt ohne und mit installierten Updates/Service-Packs auf
Delphi 7	der Fehler tritt auf (k.A. zu Updates und Service-Packs)

(keine Angabe bedeutet: mir lagen zu diesem Zeitpunkt keine Informationen vor, ob der Fehler auftritt oder nicht)

Der Work-Around

Wenn Sie eine Standard- oder Personal-Version von Delphi besitzen, haben Sie keinen Zugriff auf die Quellcodes der Unit. Mein Rat lautet in dem Fall: benutzen Sie eine möglichst aktuelle Standardversion (in der Hoffnung, dass der Fehler dort behoben wurde).
Ist das nicht der Fall, dann rufen Sie in Ihrem Programm zusätzlich den Befehl

```
InitCommonControls;
```

auf. Es spielt zwar keine Rolle, an welcher Stelle Sie den Befehl aufrufen, aber zur besseren Übersichtlichkeit empfehle ich, dass Sie den Befehl vor "`InitCommonControlsEx`" in den Quellcode eintragen. Sollte der Fehler einmal behoben worden sein, müssen Sie nicht lange suchen und können die dann überflüssige Anweisung schnell entfernen.

Der Patch

Hierfür benötigen Sie Borlands Quellcode der "`CommCtrl`"-Unit. Wenn wir uns die Deklaration am Beispiel von Delphi 5 einmal anschauen, dann finden wir in der Unit eine eigene Funktion mit dem Namen "`InitCommonControlsEx`". Dies ist kein Fehler sondern dürfte damit zu tun haben, dass die Funktion unter Windows 95 und NT4 den Internet Explorer 3 als Minimum erfordert.
Arbeitet noch jemand mit einer Originalversion dieser Systeme (ohne installiertem IE), existiert die Funktion bei ihm noch gar nicht. Der Versuch, sie dennoch (statisch) zu laden, würde eine unschöne Fehlermeldung versursachen.

Daher wird in der Unit von Borland nun versucht, die Funktion dynamisch zu laden. Zu dem Zweck wird eine zweite, interne Funktion aufgerufen, die (im Fall von D5) wie folgt aussieht:

```
procedure InitComCtl;
begin
  if ComCtl32DLL = 0 then
  begin
    ComCtl32DLL := GetModuleHandle(cctrl);
    if ComCtl32DLL <> 0 then
      @_InitCommonControlsEx := GetProcAddress(ComCtl32DLL,
'InitCommonControlsEx');
  end;
end;
```

Und hier liegt meines Erachtens nach der Fehler. In der Zeile

```
    ComCtl32DLL := GetModuleHandle(cctrl);
```

wird das Handle der geladenen DLL "`comctl32.dll`" gesucht und an die Variable übergeben. Wenn die DLL aber nicht geladen ist, erhält die Variable den Wert Null und führt in dem Fall die nachfolgenden Anweisungen auch nicht mehr aus. Fügen Sie daher nach der eben gezeigten Zeile folgendes ein:

```
    if ComCtl32DLL = 0 then ComCtl32DLL := LoadLibrary(cctrl);
```

Sollte also "`GetModuleHandle`" tatsächlich den Wert Null als Ergebnis liefern, wird die Bedingung aktiv und die DLL geladen.

3.1. Fortschrittsanzeige

3.1.1. Die Fortschrittsanzeige erzeugen

Die Fortschrittsanzeige wird allgemein zum Überbrücken gewisser Wartezeiten benutzt. Das kann das Laden einer Datei sein, das Kopieren der selben, der Download ... usw. usw. Das Control wird mit der Funktion "CreateWindowEx" erzeugt. Bei den Stilattributen verweise ich Sie wieder auf das MSDN und die Hilfe; lediglich auf dieses Attribut möchte ich näher eingehen:

Wert	Bedeutung
PBS_SMOOTH	Mit Windows 95 kam die neue Form der Fortschrittsanzeige, die den Status mit Blöcken darstellt. Standardmäßig verwendet das Control natürlich diese Anzeigeform. Mit dem links gezeigten Stilattribut können Sie allerdings die alte Form des Fortschrittsbalkens verwenden.

Im Beispielprogramm sieht der Aufruf so aus:

```
hwndProgress := CreateWindowEx(0, 'msctls_progress32', nil, WS_CHILD or
  WS_VISIBLE or PBS_SMOOTH, 10, 30, 270, 15, hWnd, 0, hInstance, nil);
```

InitCommonControlsEx

Wenn Sie den Befehl "InitCommonControlsEx" verwenden, müssen Sie für die dwICC-Membervariable die Klasse ICC_PROGRESS_CLASS benutzen.

3.1.2. Laufweite und Schritte einstellen

Standardmäßig beträgt der minimale Wert Null und der maximale Wert 100. Das sollte in den meisten Fällen ausreichend sein (zumal Sie mit "MulDiv" elegant um die evtl. lästige Prozentrechnung herumkommen ... ;o)), aber natürlich haben Sie die Möglichkeit den Arbeitsbereich der Fortschrittsanzeige selbst festzulegen. Dazu benutzen Sie die Nachricht "PBM_SETRANGE", der Sie den minimalen und maximalen Wert im zweiten Parameter übergeben. Das hört sich evtl. komplizierter aus als es in Wirklichkeit ist. Um z.B. die Laufweite auf die Werte 1 und 20 einzugrenzen, genügt dieser Befehl:

```
SendMessage(hwndProgress,PBM_SETRANGE,0,MAKELPARAM(1,20));
```

Für die Änderung der Schrittanzahl verwenden wir "PBM_SETSTEP". Standardmäßig ist die Anzahl 10, d.h. bei jedem Aufruf von "PBM_STEPIT" wird der Fortschrittsbalken um diese Anzahl erhöht. Die neue Schrittanzahl geben wir dabei im ersten Parameter an:

```
SendMessage(hwndProgress,PBM_SETSTEP,5,0);
```

3.1.3. Den Progress-Status ändern

Zum Ändern des Status der Fortschrittsanzeige stehen Ihnen zwei Nachrichten zur Verfügung. Das Beispielprogramm nutzt "PBM_STEPIT" zur schrittweisen Erhöhung der Anzeige. Hier sind keine Parameter nötig, da die Schrittanzahl ja bereits festgelegt worden ist. Die Anweisung lautet also ganz einfach nur:

```
SendMessage(hwndProgress, PBM_STEPIT, 0, 0);
```

Die zweite Variante wäre "PBM_SETPOS", wobei Sie hier die neue Position angeben. Die zuvor eingestellte Anzahl der Schritte spielt dabei keine Rolle. Die Nachricht erwartet im ersten Parameter die neue Position. Im Beispielprogramm wird auf diese Weise - beim Erreichen der höchsten Position - die Fortschrittsanzeige auf Null zurückgesetzt:

```
SendMessage(hwndProgress, PBM_SETPOS, {neue Position ->} 0, 0);
```

3.1.4. Die Farbe des Controls ändern

Eine beliebte Frage in diversen Foren ist: Wie kann ich die Farbe der Fortschrittsanzeige ändern? Der Objektinspektor von Delphi enthält bisher keine Möglichkeit, diesen Wunsch zu erfüllen. Dafür bietet das API aber die beiden Nachrichten "PBM_SETBARCOLOR" und "PBM_SETBKCOLOR".
Diese Nachrichten erwarten im lparam den neuen Farbwert, den man am einfachsten als RGB-Wert übergeben kann. Für ein dunkles Rot als Balken ist so z.B. diese Anweisung erforderlich:

```
SendMessage(hwndProgress,PBM_SETBARCOLOR,0,RGB($90,0,0));
```

Und einen schwarzen Hintergrund erhält man mit der Zeile:

```
SendMessage(hwndProgress,PBM_SETBKCOLOR,0,RGB(0,0,0));
```

Im Beispielprogramm wird dies mit Hilfe des Compilerschalter CHANGECOLOR demonstriert.

Hinweise

Bei der VCL kann, dem ersten Anschein nach, nur die Vordergrundfarbe verändert werden. Tests mit einer anderen Hintergrundfarbe verliefen leider ergebnislos. Der o.g. Codeausriss verursacht bei API-Programmen allerdings keinerlei Probleme.

Außerdem sind solche "Farbenspiele" nicht in jedem Fall empfehlenswert. Der Anwender wird sicher seine eigenen Vorstellungen bezüglich der Farben seines Systems haben, und diese sollten Sie nicht ignorieren. Es macht keinen besonders guten Eindruck, wenn sich Ihr Programm über die Einstellungen des Anwenders hinwegsetzt und eigene Farben nutzt, die u.U. im Gegensatz zu denen stehen, die der Anwender bevorzugt.

3.1.5. Der „Marquee"-Stil unter Windows XP

Unter Windows XP kennt die Fortschrittsanzeige einen neuen zusätzlichen Stil, den Sie benutzen können wenn Sie die Dauer einer Aktion nicht abschätzen können. Dabei wird die Fortschrittsanzeige nicht schrittweise erhöht, sondern eine festgelegte Anzahl an "Blöcken" bewegt sich von links nach rechts.
Um diesen Effekt zu erreichen sind allerdings die Common Controls 6.0 erforderlich. Das bedeutet, dass Sie Ihrer Anwendung eine Manifestdatei beilegen müssen. Damit ist auch klar, warum dieser Stil min. Windows XP voraussetzt.

Sie erzeugen die Fortschrittsanzeige wie am Anfang dieses Tutorials gezeigt, benutzen aber das neue Stilattribut PBS_MARQUEE:

```
hwndXP := CreateWindowEx(0,PROGRESS_CLASS,nil,
  WS_CHILD or WS_VISIBLE or PBS_MARQUEE,10,60,
  270,15,wnd,IDC_XPPROGRESS,hInstance,nil);
```

Auf die Verwendung des PBS_SMOOTH-Attributes sollten Sie aber verzichten. Der Fortschrittsbalken würde zwar ebenfalls in der neuen Form animiert werden, allerdings ist er viel zu klein:

Wenn die Themes von Windows XP aktiv sind, dann spielt die Gestaltung (ob PBS_SMOOTH oder nicht) keine Rolle, weil in dem Fall ohnehin die jeweiligen Theme-Grafiken verwendet werden. Da Sie allerdings nicht voraussetzen können, dass jeder Windows XP-Benutzer auch die Themes aktiviert hat, sollten Sie besser bei der normalen Blockansicht bleiben:

Die Animation starten und stoppen

Zum Starten der Animation steht Ihnen die neue Progressbar-Nachricht "`PBM_SETMARQUEE`" zur Verfügung, die als `wparam` den Status (**true**, **false**) und als `lparam` die Anzahl der Millisekunden für die Animation erwartet. Um den Fortschrittsbalken bspw. mit einer Geschwindigkeit von 40 Millisekunden zu bewegen, schreiben Sie:

```
SendMessage(hwndXP,PBM_SETMARQUEE,WPARAM(true),40);
```

Wenn Sie die Animation stoppen wollen, ist die Zeitangabe irrelevant. Hier ist nur wichtig, dass Sie **false** im `wparam` benutzen:

```
SendMessage(hwndXP,PBM_SETMARQUEE,WPARAM(false),0);
```

Das Beispielprogramm erzeugt eine zusätzliche Fortschrittsanzeige in diesem Stil. Voraussetzung ist, dass min. Windows XP aktiv ist, andernfalls sehen Sie lediglich eine Hinweismeldung ... :o)

3.2. Die Statuszeile

3.2.1. Die Statuszeile erzeugen

Eine Statuszeile dient gemeinhin dazu, dem Benutzer bestimmte Informationen mitzuteilen. Naturgemäß befindet sich dieses Element immer am Fuß des Fensters. Es gibt eine eigene Funktion, "CreateStatusWindow", mit der wir es erzeugen können:

```
hwndStatus := CreateStatusWindow(WS_CHILD or WS_VISIBLE or SBT_TOOLTIPS,
  nil, hWnd, IDC_STATUS);
```

Diese Funktion ist laut PSDK allerdings veraltet, so dass wir hier auch zu "CreateWindowEx" greifen können:

```
hwndStatus := CreateWindowEx(0,STATUSCLASSNAME,nil,WS_CHILD or
  WS_VISIBLE or SBT_TOOLTIPS,0,0,0,0,hWnd,IDC_STATUS,hInstance,nil);
```

Die Angaben zu Position sowie Breite und Höhe werden hierbei ignoriert.

InitCommonControlsEx

Wenn Sie den Befehl "InitCommonControlsEx" verwenden, müssen Sie für die dwICC-Membervariable die Klasse ICC_BAR_CLASSES benutzen.

3.2.2. Panels erzeugen

Da wir mehrere Panels einrichten wollen, müssen wir die Statuszeile unterteilen. Dazu haben wir die Nachricht "SB_SETPARTS". Diese Nachricht erwartet als ersten Parameter die Anzahl der Panels (max. sind 256 möglich). Im zweiten Parameter wird ein Zeiger auf ein Integerarray übergeben, das die Breite der Panels enthält. Der Wert -1 sorgt hier dafür, dass die Statuszeile die komplette Breite des Parent-Fensters ausnutzt. Als Beispiel der entsprechende Auszug aus dem Quellcode:

```
GetClientRect(hWnd, rec);
PanelWidth[0] := 40;
PanelWidth[1] := rec.Right - rec.Left - 55;
PanelWidth[2] := -1;

...

SendMessage(hwndStatus, SB_SETPARTS, 3, Integer(@PanelWidth));
```

3.2.3. Text in der Statuszeile anzeigen

Anzuzeigender Text wird mit der Nachricht "SB_SETTEXT" an das gewünschte Panel weitergegeben, wobei der Panelindex sowie ein optionaler Stil im ersten und der Textpuffer im zweiten Parameter angegeben wird. Angaben zum Stil finden Sie im MSDN und der Hilfe. Hier nur ein kurzer Auszug aus dem Beispielprogramm:

```
buffer := '10';
SendMessage(hwndStatus, SB_SETTEXT, 0, Integer(@buffer));
```

3.2.4. Tooltipps

Sie können einem Panel auch einen Tooltipp zuordnen. Dazu muss die Statuszeile mit dem Stilattribut "SBT_TOOLTIPS" erzeugt werden. Der Tipp selbst wird allerdings nur angezeigt, wenn die Beschriftung des Panels nicht vollständig zu sehen ist. Idealerweise setzen Sie also den selben Inhalt für Panel und Tooltipp, was im Beispielprogramm so aussieht:

```
buffer := 'Panel3 mit Tooltipp';
SendMessage(hwndStatus, SB_SETTEXT, 2 or SBT_NOBORDERS, Integer(@buffer));

SendMessage(hwndStatus, SB_SETTIPTEXT, 2, integer(@buffer));
```

Dazu benutzen wir die Nachricht "SB_SETTIPTEXT", deren erster Parameter den Panelindex bezeichnet, bei dem der Tooltipp später angezeigt werden soll. Dieser Text wird als Zeiger im zweiten Parameter übergeben, so wie Sie es im Codeauszug sehen können.

SB_SETTIPTEXT-Definition

```
SB_SETTIPTEXT
   wParam = (WPARAM) (int) iPart            // Panelindex
   lParam = (LPARAM) (LPCTSTR) lpszTooltip  // Zeiger auf Tooltipp
```

3.2.5. Symbole in der Statuszeile anzeigen

Als letzte Möglichkeit wollen wir noch ein Symbol in der Statuszeile einblenden. Dazu laden wir es zunächst aus den Ressourcen und benutzen dann die Nachricht "SB_SETICON", der wir wieder den Index des gewünschten Panels und das Handle des Symbols übergeben:

```
hIcon := LoadIcon(hInstance, MAKEINTRESOURCE(200));
SendMessage(hwndStatus, SB_SETICON, 1, hIcon);
```

3.3. Tooltips / Hints

3.3.1. Das Tooltipp-Fenster erzeugen

Hinter den Tooltipps verbirgt sich eigentlich nur ein Fenster mit einem eigenen Klassennamen, der vom System vorgegeben ist. Dieses Fenster übernimmt die Aufgabe, Tooltipps anzuzeigen, die Sie einem anderen Element zugeordnet haben. Das Tooltipp-Fenster erzeugen wir mit der Funktion "CreateWindowEx":

```
hToolTip := CreateWindowEx(WS_EX_TOPMOST, TOOLTIPS_CLASS, nil,
  TTS_ALWAYSTIP or TTS_NOPREFIX or WS_POPUP,
  integer(CW_USEDEFAULT), integer(CW_USEDEFAULT), integer(CW_USEDEFAULT),
  integer(CW_USEDEFAULT), wnd, 0, hInstance, nil);
```

Von großer Bedeutung ist hier der Klassenname TOOLTIPS_CLASS (eine Konstante für den tatsächlichen Namen Tooltips_Class32), der dem System signalisiert, das wir ein Tooltipp-Fenster erzeugen wollen. Als Stilattribut wählen wir:

Attribut	Bedeutung
TTS_ALWAYSTIP	Der Tooltipp erscheint auch, wenn das Hauptfenster inaktiv ist. Ohne diese Angabe müsste das Programm im Vordergrund aktiv sein, damit man den Tipp sieht.
TTS_NOPREFIX	Das Zeichen "&" wird normalerweise ausgeblendet, so dass man z.B. einen Menüeintrag auch als Tooltipp verwenden kann. Mit diesem Stilattribut kann das umgegangen werden, so dass das Kaufmanns-Und zu sehen ist.

Weitere Attribute sind im MSDN und im PSDK zu finden.

Obwohl wir beim Erzeugen des Tooltipp-Fensters bereits das Attribut "WS_EX_TOPMOST" benutzt haben, legen wir es mit folgendem Aufruf noch einmal als "oberstes Fenster" fest:

```
SetWindowPos(hToolTip, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE or
  SWP_NOSIZE or SWP_NOACTIVATE);
```

InitCommonControlsEx

Wenn Sie den Befehl "InitCommonControlsEx" verwenden, müssen Sie für die dwICC-Membervariable die Klasse ICC_BAR_CLASSES benutzen.

3.3.2. Tooltipps registrieren

Unser Tooltipp-Fenster kann glücklicherweise mehr als einen Tooltipp beinhalten. Es ist also nicht erforderlich, für jeden Tipp ein eigenes Fenster zu erzeugen. Wenn wir einen Tooltipp registrieren wollen, benötigen wir die Nachricht "TTM_ADDTOOL", der wir als lParam einen Zeiger auf das TToolInfo-Record übergeben.

TToolInfo-Definition zeigen

```
typedef struct tagTOOLINFO{
    UINT      cbSize;       // Größe des Records
    UINT      uFlags;       // Flags
    HWND      hwnd;         // Handle des Fensters,
                            // das den Tipp zeigen soll
    UINT_PTR  uId;          // ID
    RECT      rect;         // Fläche des Fensters
    HINSTANCE hinst;        // Anwendungsinstanz
    LPTSTR    lpszText;     // Tipp-Text
#if (_WIN32_IE >= 0x0300)
    LPARAM lParam;
#endif
}
```

Wie in solchen Fällen üblich, müssen wir vor dem Aufruf erst einmal die Größe des Records festlegen:

```
ti.cbSize   := sizeof(TToolInfo);
```

Des Weiteren benötigen wir das Handle des Elements, dem wir den Tooltipp zuordnen wollen. Und weil unser Tipp auch an der richtigen Stelle erscheinen soll, benötigen wir außerdem noch die Maße des jeweiligen Elementes:

```
// Fenster-Handle (= Control)
ti.hwnd     := wnd;
ti.uId      := wnd;

// Maße des Fensters
GetClientRect(wnd,ti.Rect);
```

Ein Wort vielleicht noch zu den Flags:

```
ti.uFlags   := TTF_SUBCLASS or TTF_IDISHWND;
```

Flag	Bedeutung
TTF_SUBCLASS	Maus-bezogene Nachrichten werden automatisch an das Tooltipp-Fenster weitergeleitet
TTF_IDISHWND	Die Wert von "`uID`" entspricht dem Fenster-Handle

Hinweis

Da wir hier typische Controls (Buttons, Eingabefelder usw.) verwenden, können wir auf das Bearbeiten der Maus-bezogenen Nachrichten verzichten. Diese werden durch das Flag "`TTF_SUBCLASS`" automatisch vom Control an das Tooltipp-Fenster weitergeleitet.

Das funktioniert (laut PSDK) nicht, wenn ein Programm z.B. ein System-definiertes Fenster nutzt. In einem solchen Fall werden die Nachrichten an dieses System-Fenster und nicht an das Programm geschickt. Man müsste also nicht nur die Nachrichten abfangen, es wäre auch erforderlich, die Anzeige und Positionierung der Tooltipps selbst zu übernehmen.
Da es aber in den meisten Fällen darum gehen dürfte, Tooltipps für die Controls des selben Programms einzublenden, sparen Sie mit dem o.g. Flag eine Menge Arbeit.

Das ganze übergeben wir dann mit der schon genannten Nachricht an das Tooltipp-Fenster:

```
SendMessage(hToolTip,TTM_ADDTOOL,0,LPARAM(@ti));
```

TTM_ADDTOOL-Definition zeigen

```
TTM_ADDTOOL
  (WPARAM) wParam        // nicht benutzt; muss Null sein
  (LPARAM) lParam        // Zeiger auf "TToolInfo"-Record
```

Damit wir diesen Code nicht für jedes Element schreiben müssen, bietet sich eine eigene Funktion oder Prozedur an, die wir aufrufen können. Sie finden sie im Beispielprogramm als "AddToolTip". Der Prozedur übergeben Sie das Handle des gewünschten Controls, die Anwendungsinstanz (in dem Fall also immer hInstance) und natürlich den Text, den Sie als Tooltipp sehen wollen, und der im Beispiel als Konstante deklariert ist:

```
AddToolTip(hEdit,hInstance,TIPP_EDIT);
```

Hinweis

Wenn wir mit Dialog-Ressourcen arbeiten, müssen wir ein klein wenig anders vorgehen. Da wir nur das Handle unseres Dialogs haben und lediglich über eine ID auf das gewünschte Element zugreifen können, verwenden wir die Funktion "GetDlgItem", um an das Handle des Elements heranzukommen. Als Beispiel gehen wir davon aus, dass das Dialog-Handle der Variablen "hDlgWnd" zugeordnet ist, und dass unser Eingabefeld die ID "120" hat. Der Aufruf sieht dann so aus:

```
AddToolTip(GetDlgItem(hDlgWnd,120),hInstance,TIPP_EDIT);
```

3.3.3. Den Text eines Tooltipps ändern

In unserem Beispiel ist diese Funktion mehr oder weniger nur Spielerei. Es gibt aber durchaus reale Anwendungsgebiete, in denen das Ändern des Tipp-Textes sinnvoll ist. So könnte man z.B. die Beschreibungen von Buttons, Eingabefeldern usw. abhängig von irgendwelchen Bedingungen ändern und so den Anwender stets auf dem Laufenden halten. usw. Zur Änderung des Textes ist natürlich ein neuer Text erforderlich (selbstverständlich!). Ebenso muss aber auch das Handle des entsprechenden Fensters (Elements) angegeben werden, damit auch der richtige Tooltipp geändert wird und der neue Text auch über richtigen Control erscheint. Aktualisiert wird der Tooltipp dann mit der Nachricht "TTM_UPDATETIPTEXT":

```
SendMessage(hToolTip,TTM_UPDATETIPTEXT,0,integer(@ti));
```

TTM_UPDATETIPTEXT-Definition zeigen

```
TTM_UPDATETIPTEXT
  (WPARAM) wParam        // nicht benutzt; muss Null sein
  (LPARAM) lParam        // Zeiger auf "TToolInfo"-Record
```

Sie können diese Funktion ausprobieren, indem Sie im Beispielprogramm einen neuen Text in das Eingabefeld schreiben und dann den Button "Tipp ändern" benutzen. Daraufhin zeigt Ihnen der "Schließen"-Button den von Ihnen geschriebenen Text an. Das Programm liest dazu den Text aus dem Eingabefeld aus und übergibt ihn an die o.g. Prozedur. - Wie gesagt: in dem Fall ist es Spielerei. :o)

3.3.4. Tooltipps aktivieren und deaktivieren

Tooltipps lassen sich auch während des Programmlaufes vorübergehend aus- und wieder einschalten. Das gilt aber generell für alle Tooltipps. Das Beispiel demonstriert dies mit Hilfe einer Checkbox. Standardmäßig sind die Tipps aktiviert. Wenn Sie das Häkchen aus der Checkbox entfernen, werden sie deaktiviert. Zu diesem Zweck nutzen wir die Nachricht "TTM_ACTIVATE".
Als WPARAM geben wir **true** an, wenn die Tipps aktiviert sein sollen; und **false**, wenn sie deaktiviert sein sollen. Die Prüfung unserer Checkbox lautet daher also:

```
bFlag := SendMessage(hCheckBox,BM_GETCHECK,0,0) = BST_CHECKED;
```

Den so ausgelesenen Status senden wir mit Hilfe der o.g. Nachricht an unser Tooltipp-Fenster:

```
SendMessage(hToolTip,TTM_ACTIVATE,integer(bFlag),0);
```

TTM_ACTIVATE-Definition

```
TTM_ACTIVATE
  (WPARAM) wParam        // fActivate
  (LPARAM) lParam        // nicht benutzt; muss Null sein
```

3.3.5. Tooltipps entfernen

Wenn Sie einen Tooltipp dauerhaft entfernen wollen, dann verwenden Sie dazu die Nachricht "`TTM_DELTOOL`". Das `TToolInfo`-Record erwartet von Ihnen die gültige Angabe des Control- bzw. Fenster-Handles, dessen Tipp Sie entfernen wollen. Das Beispielprogramm verwendet diese Funktion nicht, aber eine Aufrufmöglichkeit könnte so aussehen:

```
// Strukturgröße festlegen
ti.cbSize   := sizeof(TToolInfo);

// das Handle des Eingabefeldes,
// & die Anwendungsinstanz
ti.hwnd     := hEdit;
ti.hInst    := hInstance;

// Tooltipp entfernen
SendMessage(hToolTip,TTM_DELTOOL,0,integer(@ti));
```

Alle anderen Tooltipps bleiben davon unberührt. Es würde tatsächlich nur der Tipp unseres Eingabefeldes entfernt werden.

TTM_DELTOOL-Definition zeigen

```
TTM_DELTOOL
  (WPARAM) wParam        // nicht benutzt; muss Null sein
  (LPARAM) lParam        // Zeiger auf "TToolInfo"-Record
```

3.3.6. Weitere Möglichkeiten für Tooltipps

Neben den vorgestellten Funktionen gibt es noch weitere Dinge, die man mit den Tooltipps machen kann. So lässt sich z.B. die Farbe des Tipps unabhängig von den Einstellungen des Systems ändern. Des Weiteren lässt sich einstellen, nach wie vielen Millisekunden ein Tipp erscheinen soll, wie lange er sichtbar bleiben soll, und wieviel Zeit zwischen dem Anzeigen verschiedener Tipps vergehen soll, wenn man den Mauszeiger von einem Control zum anderen bewegt. - usw. usw.

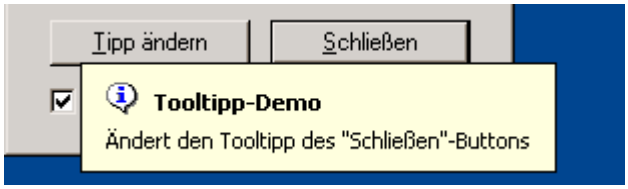
Ein paar neuere Funktionen sind allerdings für Besitzer älterer Delphi-Versionen nicht verfügbar. Die Units von Borland sind in dem Fall schlichtweg veraltet.

Am einfachsten ist es noch, wenn man nur ein neues Attribut ausprobieren möchte. Als Beispiel wollen wir die Tooltipps im Cartoon-Stil (Balloon) anzeigen lassen. Dazu brauchen wir lediglich eine aktuelle Version der Headerdatei "`commctrl.h`" (PSDK, MSDN), aus der wir uns die entsprechende Konstante herausschreiben:

```
const
  TTS_BALLOON = $40;
```

Diese können wir nun nutzen, und sofern unsere DLL aktuell genug ist, erscheint der Tipp im neuen Design:

Ein bisschen problematischer wird es, wenn Sie auf alle Neuheiten zurückgreifen wollen. In dem Fall sollten Sie die Unit "`CommCtrl.pas`" anpassen. Auch dazu benötigen Sie als Referenz eine aktuelle Version der o.g. Headerdatei. Und das Ergebnis spricht dann für sich:



Hinweis

Das Beispielprogramm nutzt eine spezielle Unit, "`CommCtrl_Fragment.pas`", die die notwendigen Erweiterungen dieses Kapitels enthält. Durch einen speziellen Compilerschalter wird die Unit aber nur mit Delphi 5 verwendet.
Prüfen Sie bitte vorher, ob die Ergänzungen bei Ihrer Delphi-Version überhaupt noch notwendig sind. Probieren Sie dann, ob sich das Programm mit Ihrer Delphi-Version und der genannten Unit kompilieren lässt.

3.3.7. Die "CommCtrl.pas" ergänzen

Die Ergänzungen sind hauptsächlich neue Konstantenwerte, die wir aus einer aktuellen Version der Headerdatei "`commctrl.h`" übernehmen können:

```
const
  {$EXTERNALSYM TTS_NOANIMATE}
  TTS_NOANIMATE           = $10;
  {$EXTERNALSYM TTS_NOFADE}
  TTS_NOFADE              = $20;
  {$EXTERNALSYM TTS_BALLOON}
  TTS_BALLOON             = $40;
  {$EXTERNALSYM TTS_CLOSE}
  TTS_CLOSE               = $80;

  // ToolTip Icons (Set with TTM_SETTITLE)
  {$EXTERNALSYM TTI_NONE}
  TTI_NONE                = 0;
  {$EXTERNALSYM TTI_INFO}
  TTI_INFO                = 1;
  {$EXTERNALSYM TTI_WARNING}
  TTI_WARNING             = 2;
  {$EXTERNALSYM TTI_ERROR}
  TTI_ERROR               = 3;

  {$EXTERNALSYM TTF_PARSELINKS}
  TTF_PARSELINKS          = $1000;


  {$EXTERNALSYM TTM_GETBUBBLESIZE}
  TTM_GETBUBBLESIZE        = WM_USER + 30;
  {$EXTERNALSYM TTM_ADJUSTRECT}
  TTM_ADJUSTRECT           = WM_USER + 31;
  {$EXTERNALSYM TTM_SETTITLEA}
  TTM_SETTITLEA            = WM_USER + 32;
  {$EXTERNALSYM TTM_SETTITLEW}
  TTM_SETTITLEW            = WM_USER + 33;


  {$EXTERNALSYM TTM_POPUP}
  TTM_POPUP                = WM_USER + 34;
  {$EXTERNALSYM TTM_GETTITLE}
  TTM_GETTITLE             = WM_USER + 35;


type
  _TGETTITLE = packed record
    dwSize : DWORD;
    uTitleBitmap : UINT;
    cch : UINT;
    pszTitle : PWideChar;
  end;
  TGetTitle = _TGETTITLE;
  PGetTitle = ^TGetTitle;


const
  {$EXTERNALSYM TTM_SETWINDOWTHEME}
  TTM_SETWINDOWTHEME      = CCM_SETWINDOWTHEME;
```

Um Probleme mit der letzten Konstante zu vermeiden, ist weiter am Anfang der Unit noch `CCM_SETWINDOWTHEME` zu deklarieren (suchen Sie zum Einfügen bitte nach `INFOTIPSIZE`). Sie sollten aber bedenken, dass diese Konstanten nur unter Windows XP benutzt werden können:

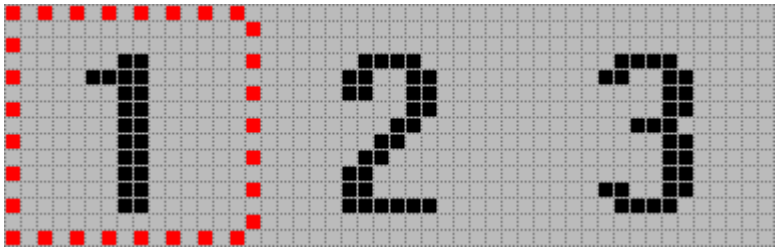
```
const
  {$EXTERNALSYM CCM_SETVERSION}
  CCM_SETVERSION          = CCM_FIRST + $07;
  {$EXTERNALSYM CCM_GETVERSION}
  CCM_GETVERSION          = CCM_FIRST + $08;
  {$EXTERNALSYM CCM_SETNOTIFYWINDOW}
  CCM_SETNOTIFYWINDOW     = CCM_FIRST + $09; // wParam == hwndParent.
  {$EXTERNALSYM CCM_SETWINDOWTHEME}
  CCM_SETWINDOWTHEME      = CCM_FIRST + $0b;
  {$EXTERNALSYM CCM_DPISCALE}
  CCM_DPISCALE            = CCM_FIRST + $0c; // wParam == Awareness
```

3.4. Die Toolbar

3.4.1. Grundlagen zu den Bitmaps einer Toolbar

Beginnen wir zunächst mit den Bitmaps für unsere Toolbar. Es hat den Anschein, als wären es verschiedene Dateien, die den Toolbar-Buttons zugeordnet sind. Tatsächlich handelt es sich aber nur um eine einzige Bitmap. Wenn Sie die Grafiken nicht mit dem VisualStudio-Editor von Microsoft erstellen, dann achten Sie bitte darauf, dass alle Einzelbilder die selbe Größe haben, und dass die Bitmap dies entsprechend berücksichtigt.

Nehmen wir als Beispiel an, Sie benötigen drei Grafiken mit den Abmessungen 16x15. Dann sollte Ihre Bitmap eine Größe von 48x15 Pixeln haben. Kopieren Sie dann die drei Einzelgrafiken nahtlos nebeneinander und speichern Sie die Bitmap. Zum besseren Verständnis habe ich die Bitmap des Beispielprogramms vergrößert und den Bereich des ersten Buttons umrahmt:



Einfacher geht es mit dem schon erwähnten VS-Editor. Hier können Sie - wie vom Zeichenprogramm gewohnt - arbeiten, und der Editor sorgt im Hintergrund dafür, dass die einzelnen Toolbutton-Grafiken in einer einzigen Bitmap gespeichert werden.

3.4.2. Die Toolbar-Buttons erzeugen

Für unsere Buttons ist ein `TBBUTTON`-Record erforderlich. Es enthält verschiedene Eigenschaften, und Ihre Aufgabe besteht eigentlich nur darin, die Anzahl der Buttons festzulegen, deren Eigenschaften Sie dann im Vorfeld natürlich auch schon füllen könnten. In unserem Beispiel sind vier Buttons deklariert worden:

```
var
  tbButtons: Array[0..3] of TTBButton;
```

TBBUTTON-Definition

```
typedef struct _TBBUTTON {
    int             iBitmap;
    int         idCommand;
    BYTE             fsState;
    BYTE             fsStyle;
#ifdef _WIN64
    BYTE              bReserved[6]    // padding for alignment
#elif defined(_WIN32)
    BYTE             bReserved[2]    // padding for alignment
#endif
    DWORD_PTR  dwData;
    INT_PTR    iString;
}
```

Ein typischer Toolbar-Button wird dann so erzeugt:

```
with tbButtons[0] do begin
  iBitmap := 0;                        // Bitmap-Index (Null-basierend)
  idCommand := IDC_BUTTON1;            // Button-ID
  fsState := TBSTATE_ENABLED;          // Button-Status
  fsStyle := BTNS_BUTTON;              // Button-Typ
  dwData := 0;
  iString := 0;
end;
```

Für einen Separator nutzen Sie ein anderes Stilattribut, während Sie den Großteil der anderen Parameter auf Null setzen können (weil diese bei einem Separator ja nicht benötigt werden):

```
with tbButtons[1] do begin
  iBitmap := 0;
  idCommand := 0;
  fsState := TBSTATE_ENABLED;
  fsStyle := BTNS_SEP;                  // Separator-Stilattribut
  dwData := 0;
  iString := -1;
end;
```

Ein drittes Stilattribut finden Sie im dritten Button (eigentlich ist es der vierte, weil man den Separator mitrechnen muss), der später ein Dropdown-Menü anzeigen soll:

```
with tbButtons[3] do begin
  ...
  fsStyle := BTNS_DROPDOWN;             // Dropdown-Stil
  ...
end;
```

Hinweis

Das Beispielprogramm benutzt eine spezielle Unit, "`CommCtrl_Fragment.pas`". Diese Unit enthält die neuen Button-Stile der Version 5.80, die von Microsoft als neuer Standard empfohlen werden. Diese Stilattribute beginnen mit dem Präfix "`BTNS_`", entsprechen aber ansonsten den bisher gewohnten Attributen. Hintergedanke ist, Verwechslungen mit den Fensterattributen der Toolbar zu vermeiden, die mit dem Präfix "`TBSTYLE_`" beginnen. Das Beispielprogramm demonstriert Ihnen die Anwendung der neuen Stile.
Die Unit wird allerdings nur benutzt, wenn Sie das Beispielprogramm mit Delphi 5 kompilieren. Sie sollten vorher prüfen, ob Ihre Delphi-Version diese Änderungen überhaupt noch benötigt. Probieren Sie dann ggf. aus, ob Sie das Programm mit der genannten Version kompilieren können.

3.4.3. Die Toolbar erzeugen

Jetzt können wir die Toolbar erzeugen. Obwohl - damit auch alles funktioniert, müssen wir zuvor noch die Bitmap auswählen und laden. Dazu benutzen wir die Funktion "`CreateMappedBitmap`", die anhand unserer Ressourcenbitmap eine Bitmap für unsere Toolbar erzeugt:

```
hBitmap := CreateMappedBitmap(hInstance, 100, 0, nil, 0);
```

CreateMappedBitmap-Definition

```
HBITMAP CreateMappedBitmap(
    HINSTANCE hInstance,      // Anwendungsinstanz
    int idBitmap,             // Ressourcen-ID der Bitmap
    UINT wFlags,              // Bitmapflag (Null, oder CMB_MASKED)
    LPCOLORMAP lpColorMap,    // Colormap mit Farbinfos zum Mappen der Bitmap
    int iNumMaps              // Anzahl der benutzen Colormaps (oder nil)
);
```

Wenn das erledigt ist, können wir mit "CreateToolbarEx" die Toolbar erzeugen:

```
hToolbar := CreateToolbarex(hWnd, WS_CHILD or WS_VISIBLE or
  TBSTYLE_TOOLTIPS or CCS_NODIVIDER or TBSTYLE_FLAT, IDC_TOOLBAR, 3, 0,
  hBitmap, @tbButtons, length(tbButtons), 0, 0, 16, 15, sizeof(TTBBUTTON));
```

CreateToolbarEx-Definition

```
HWND CreateToolbarEx(
    HWND hwnd,                // Handle des übergeordneten Fensters
    DWORD ws,                 // Stil der Toolbar
    UINT wID,                 // Toolbar-ID
    int nBitmaps,             // Anzahl der Bitmaps (Null-basierend)
    HINSTANCE hBMInst,        // Modulinstanz der Datei, die die
                              // Bitmaps enthält; nur nötig bei
                              // Systembitmaps - ansonsten Null
    UINT wBMID,               // Ressourcen-ID der Bitmap, bzw.
                              // Handle auf eine Bitmap-Ressource
    LPCTBBUTTON lpButtons,    // Zeiger auf das TBBUTTON-Record
    int iNumButtons,          // Anzahl der Buttons
                              // (ACHTUNG! Separatoren mitzählen!)
    int dxButton,
    int dyButton,             // Breite und Höhe der Buttons
    int dxBitmap,
    int dyBitmap,             // Breite und Höhe der Bitmap
    UINT uStructSize          // Größe des TBBUTTON-Records
);
```

Hinweis

Im PSDK wird die Funktion "CreateToolbarEx" als veraltet bezeichnet. Stattdessen empfiehlt Ihnen Microsoft die Verwendung von "CreateWindowEx". Das erfordert aber ein paar Zeilen mehr Code. Zuerst erzeugen Sie die Toolbar, wobei Sie Angaben zu Position sowie Höhe und Breite ignorieren können:

```
hToolBar := CreateWindowEx(0,TOOLBARCLASSNAME,nil,WS_CHILD or
  WS_VISIBLE or CCS_NODIVIDER or TBSTYLE_TOOLTIPS or TBSTYLE_FLAT
  {$IFDEF RIGHTTEXT} or TBSTYLE_LIST {$ENDIF},
  0,0,0,0,hWnd,IDC_TOOLBAR,hInstance,nil);
```

Da die "CreateWindowEx"-Funktion für viele Zwecke benutzt wird, müssen Sie dem neu erstellten Fenster nun auch die Größe des TTBBUTTON-Records übermitteln und dann natürlich auch die Buttons selbst:

```
SendMessage(hToolBar,TB_BUTTONSTRUCTSIZE,sizeof(TTBBUTTON),0);
SendMessage(hToolBar,TB_ADDBUTTONS,length(tbButtons),LPARAM(@tbButtons));
```

Und für die Bitmap benötigen Sie nun eine Variable vom Typ TTBAddBitmap, wobei Sie vorher natürlich noch die Bitmap laden müssen:

```
hBitmap    := CreateMappedBitmap(hInstance, 100, 0, nil, 0);
aBmp.hInst := 0;
aBmp.nID   := hBitmap;
SendMessage(hToolBar,TB_ADDBITMAP,3,LPARAM(@aBmp));
```

Wenn Sie eine Farbe der Grafik transparent darstellen wollen, dann benötigen Sie ein Record vom Typ `TColorMap`. Wenn Sie mehr als eine Farbe ändern wollen, dann sollten Sie ein mehrere `TColorMap`-Records zu einem Array verbinden.
Das Beispielprogramm enthält eine zweite Bitmapressource, bei der jeder Button einen anders farbigen Hintergrund hat. Diese drei Farben sollen nun durch die typische Hintergrundfarbe für 3D-Objekte ersetzt werden. Auch die Farbe der Beschriftung soll dem System angepasst werden. Dazu benötigen wir, wie schon gesagt, ein Array aus `TColorMap`-Records:

```
var
  cm : array[0..3]of TColorMap;
```

Jedem Record muss die Farbe zugewiesen werden, die ersetzt werden soll. Und natürlich muss auch die gewünschte neue Farbe angegeben werden. Am Beispiel von Rot würde das so aussehen:

```
cm[0].cFrom := $000000ff;
cm[0].cTo   := GetSysColor(COLOR_3DFACE);
```

Das gleiche machen Sie mit Blau bzw. mit allen Farben, die Sie ersetzen wollen. Dann ändern Sie den o.g. Aufruf von "`CreateMappedBitmap`" so ab, dass Sie einen Zeiger auf das Array und die Anzahl der zu ersetzenden Farben angeben:

```
hBitmap     := CreateMappedBitmap(hInstance,200,0,@cm[0],length(cm));
```

Damit werden die Hintergrundfarben (Rot, Blau, Lila) ersetzt, und die Grafik bzw. die Toolbar-Buttons sollten eigentlich vollkommen normal aussehen. Wenn Sie das Beispiel nachvollziehen wollen, dann entfernen Sie in der Demo bitte den Punkt bei diesem Compilerschalter

```
{.$DEFINE USEMASKBITMAP}
```

Damit der Pfeil auf dem dritten Button (Dropdown-Stil) korrekt angezeigt wird, senden wir die Nachricht "`TB_SETEXTENDEDSTYLE`" mit der Anweisung, den Pfeil zu zeichnen, an die Toolbar:

```
SendMessage(hToolBar, TB_SETEXTENDEDSTYLE, 0, TBSTYLE_EX_DRAWDDARROWS);
```

Wenn Sie Text auf den Buttons anzeigen wollen, benötigen Sie String, der alle Buttonbeschriftungen enthält, die jeweils durch das Zeichen `#0` voneinander getrennt sind. Die letzte Beschriftung wird mit zwei `#0`-Zeichen abgeschlossen:

```
var
  TB_Text : string = 'Button 1'#0'Button 2'#0'Button 3'#0#0;
```

Diesen Text übergeben Sie dann mit Hilfe der Nachricht "`TB_ADDSTRING`" an die Toolbar. Da es sich hierbei um einen String handelt, ist das erste Zeichen anzugeben:

```
SendMessage(hToolBar,TB_ADDSTRING,0,LPARAM(@TB_Text[1]));
```

Alternativ dazu ginge auch "`pchar(TB_Text)`" bzw. "`pointer(TB_Text)`". Welcher Button welche Beschriftung besitzt, das entscheiden Sie beim Erzeugen der Buttons mit Hilfe der Membervariablen "`iString`".
Und damit besitzt unser Programm nun seine Toolbar.

InitCommonControlsEx

Wenn Sie den Befehl "`InitCommonControlsEx`" verwenden, müssen Sie für die `dwICC`-Membervariable die Klasse `ICC_BAR_CLASSES` benutzen.

3.4.4. Bitmaps aus dem System

Wir hatten ja schon besprochen, wie man Bitmaps für Toolbars erstellt. Hier soll nun demonstriert werden, wie man vorhandene Grafiken des Systems verwenden kann. Betrachten wir dazu einmal den Zweck von typischen Toolbars: die meisten Programme bieten damit einen schnelleren Zugriff auf bekannte Menübefehle, wie etwa Öffnen, Speichern, Kopieren, Ausschneiden, usw. Für solche doch recht gebräuchlichen Befehle stellt Windows bereits fertige Grafiken zur Verfügung. Das spart zum einen Ressourcen und vermindert auf der anderen Seite den Arbeitsaufwand. Allerdings ist man dafür in der Auswahl natürlich eingeschränkt.

3.4.4.1. Vorbereitungen

Wie bereits hier beschrieben, bereiten wir unser `TTBButton`-Record vor. Nur diesmal benutzen wir in der `iBitmap`-Membervariablen nicht die ID einer externen Bitmap! Stattdessen verwenden wir Konstanten, die wir in der Unit "`CommCtrl.pas`" finden können, und die mit dem Präfix "`STD_`" beginnen. Ein paar der bekanntesten sollen hier aufgezählt werden:

Konstante	Beschreibung
STD_COPY	entspricht der Bitmap "kopieren"
STD_CUT	entspricht der Bitmap "ausschneiden"
STD_DELETE	entspricht der Bitmap "löschen"
STD_FILENEW	entspricht "Datei/Neu"
STD_FILEOPEN	entspricht "Datei/Öffnen"
STD_FILESAVE	entspricht "Datei/Speichern"

Daneben gibt es noch andere Bitmapkonstanten, die mit dem Präfix "`VIEW_`" beginnen und aus dem Windows-Explorer bekannt sein dürften. Etwa:

Konstante	Beschreibung
VIEW_DETAILS	entspricht "Ansicht/Details"
VIEW_LARGEICONS	entspricht "Ansicht/große Symbole"
VIEW_SMALLICONS	entspricht "Ansicht/kleine Symbole"

(da die Bitmaps systembedingt abweichen könnten, spare ich mir eine grafische Darstellung)

3.4.4.2. Anwendung

Ein Button mit einer solchen Systembitmap könnte also wie folgt deklariert werden:

```
with tbButtons[0] do
  begin
    iBitmap := STD_FILEOPEN;

    { ... }
  end;
```

Nun können wir unsere Toolbar erstellen, wobei wir aber so ziemlich alle Parameter (also Größe der Buttons, Anzahl der Bitmaps, usw.) vernachlässigen können. Da wir damit zwar die Toolbar nicht aber die Bitmaps haben, benötigen wir als nächstes eine Variable vom Typ `TBAddBitmap`. Diese Variable haben Sie bereits hier kennengelernt, als der neue, von Microsoft empfohlene Weg zum Erstellen einer Toolbar demonstriert wurde.
In diesem Fall übergeben wir der Membervariablen `hInst` den Wert "`HINST_COMMCTRL`". Damit ist schon festgelegt, dass wir die Systembitmaps nutzen wollen. Allerdings müssen wir noch angeben, welche exakt das sein sollen. Im Fall des eben gezeigten Button-Beispiels interessieren uns die Standardbitmaps ("`STD_`"-Präfix), so dass wir die Wahl zwischen "`IDB_STD_LARGE_COLOR`" (Standardgrafiken, groß) und "`IDB_STD_SMALL_COLOR`" (Standardgrafiken, klein) haben.

```
aBmp.hInst := HINST_COMMCTRL;
aBmp.nID := IDB_STD_SMALL_COLOR;
```

Für die View-Bitmaps lauten die Konstanten entsprechend "IDB_VIEW_LARGE_COLOR" und "IDB_VIEW_SMALL_COLOR".

TBADDBITMAP-Definition

```
typedef struct {
    HINSTANCE hInst;
    UINT_PTR nID;
  } TBADDBITMAP
```

Der Rest ist recht einfach - Briefmarke drauf und das ganze an die Toolbar schicken :o)

```
SendMessage(hToolbar,TB_ADDBITMAP,0,LPARAM(@aBmp));
```

Zur Demonstration liegt ein kleines, separates Beispielprogramm bei, das die notwendigen Schritte anhand einer Beispieltoolbar zeigt.

3.4.4.3. Die VCL-Toolbar und die Systembitmaps

Auch mit der **Toolbar**-Komponente der VCL können Sie die Systembitmaps nutzen. Dazu erzeugen Sie zunächst die gewünschten Buttons.
Um die Bitmaps zuzuweisen, gibt es zwei Möglichkeiten:

1. Wenn Sie die Eigenschaft "**ImageIndex**" im Objektinspektor nutzen wollen, dann benötigen Sie numerischen Werte der o.g. Konstanten, etwa von STD_FILENEW, weil Sie diese ja nicht direkt angeben können. Sie finden die Werte in der Unit "CommCtrl.pas" bzw. in der Headerdatei "CommCtrl.h" (PSDK).
2. Etwas aufwändiger, dafür aber (in Bezug auf evtl. Änderungen) sicherer, ist der Weg, die Bitmapwerte im "**OnCreate**"-Ereignis der Form zuzuweisen. In dem Fall können Sie auch die o.g. Konstanten nutzen:

   ```
   Toolbar1.Buttons[0].ImageIndex := STD_FILENEW;
   ```

Die Bitmaps selbst werden wie oben beschrieben geladen. Es bietet sich an, den notwendigen Code im "**OnCreate**"-Ereignis der Form unterzubringen.

3.4.5. Das Klickereignis von Toolbar-Buttons

Wie bei normalen Buttons benutzen wir auch bei Toolbar-Buttons die Nachricht "WM_COMMAND" und prüfen ob der Benachrichtungscode "BN_CLICKED" ist:

```
WM_COMMAND:
  begin
    if hiword(wParam) = BN_CLICKED then
      case loword(wParam) of
        IDC_BUTTON1:
          SendMessage(hwnd, WM_CLOSE, 0, 0);
        IDC_BUTTON2,
        IDC_BUTTON3:
          MessageBox(hWnd, 'Button angeklickt', 'Kuckuck', MB_ICONINFORMATION);
      end;
  end;
```

Sie sehen, dass das Prinzip mit dem Abfangen von normalen Buttonklicks identisch ist.

3.4.6. Tooltipps und Dropdown-Menüs

Die bekannten Toolbars haben unserer im Moment nur noch zwei Dinge voraus: zum einen fehlen bei uns die Tooltipps, die beim Platzieren des Mauszeigers auf einem Button erscheinen, und zum anderen müssen wir noch unser Menü erzeugen, das beim dritten Button (im Dropdown-Stil) angezeigt werden soll.

Dazu nutzen wir die Nachricht "WM_NOTIFY". Diese Nachricht wird von einem Control an das übergeordnete Fenster gesendet, sobald irgendein Ereignis eingetreten ist. Als Benachrichtigungscodes interessieren uns hier "TTN_NEEDTEXT" für die Tooltipps:

```
case PNMToolBar(lParam)^.hdr.code of
  TTN_NEEDTEXT:
    case PToolTipText(lParam).hdr.idFrom of
      IDC_BUTTON1: PToolTipText(lParam).lpszText := 'ToolbarButton 1';
      IDC_BUTTON2: PToolTipText(lParam).lpszText := 'ToolbarButton 2';
      IDC_BUTTON3: PToolTipText(lParam).lpszText := 'ToolbarButton 1';
    end;
end;
```

und "TBN_DROPDOWN" für das Menü, wobei wir die Position des Buttons direkt mit Hilfe des lParam ermitteln. Das bietet den Vorteil, dass wir die Buttons beliebig neu anordnen können - das Menü erscheint dann trotzdem unter dem richtigen:

```
case PNMToolBar(lParam)^.hdr.code of
  TBN_DROPDOWN:
    begin
      { wo ist der 3. Button }
      SendMessage(hToolbar,TB_GETRECT,PNMToolBar(lParam)^.iItem,LPARAM(@Rect));
      pt.x := Rect.Left;
      pt.y := Rect.Bottom + 2;
      ClientToScreen(hWnd, pt);

      { Popupmenü erzeugen - s. Menü-Tutorial }
    end;
```

3.4.7. Die Toolbar anpassen

Im letzten Kapitel wollen wir uns noch anschauen, wie man die Buttons einer Toolbar beliebig anordnen kann und wie diese Änderungen dann entsprechend gespeichert und geladen werden.
Die VCL von Delphi unterstützt diese Funktionalität leider erst ab Version 7, obwohl sie (rein technisch) vom Betriebssystem bereitgestellt wird. Daher können wir sie für unser kleines nonVCL-Beispiel auch mit älteren Delphi-Versionen problemlos verwenden.

3.4.7.1. Änderungen im Programm

Zuerst einmal müssen wir die Toolbar mit dem zusätzlichen Stilattribut CCS_ADJUSTABLE erstellen:

```
hToolBar := CreateWindowEx(0,TOOLBARCLASSNAME,nil,WS_CHILD or WS_VISIBLE or
  CCS_ADJUSTABLE or CCS_NODIVIDER or TBSTYLE_TOOLTIPS or TBSTYLE_FLAT or
TBSTYLE_LIST,
  0,0,0,0,hWnd,IDC_TOOLBAR,hInstance,nil);
```

Dieses Attribut reicht schon aus, um die Buttons mit gedrückter Shift-Taste zu verschieben. Nur nutzt das natürlich bis jetzt noch nichts. Und wenn Sie testweise einen Doppelklick in einem freien Bereich der Toolbar ausführen, dann werden Sie evtl. ein kurzes Flackern bemerken. Dieses Flackern ist die Dialogbox, auf die wir später noch eingehen werden. Dass sie nicht erscheint bzw. sichtbar bleibt, hat natürlich auch seine Gründe.

Apropos: Drag & Drop. Wenn Sie die Buttons lieber mit gedrückter ALT-Taste umsortieren wollen, dann verwenden Sie zusätzlich noch das Attribut TBSTYLE_ALTDRAG beim Erzeugen der Toolbar.

Zu guter Letzt sollten Sie noch sicherstellen, dass die Variable, die Sie für Ihre Toolbar-Buttons benutzt haben (in dem Fall "tbButtons"), eine globale Variable ist. Eine lokale Variable in Ihrer "WndProc" würde nur Probleme verursachen.

3.4.7.2. Welche Buttons dürfen manipuliert werden?

Schauen wir uns im PSDK einmal an, was beim Versuch, die Toolbar zu ändern, geschieht:

1. Die Toolbar sendet eine TBN_BEGINADJUST-Benachrichtigung, die uns aber nicht weiter interessiert.
2. Es schließt sich die TBN_INITCUSTOMIZE-Benachrichtigung an, die wir verwenden können, um den Hilfe-Button aus dem Dialogfeld (auf das wir noch warten) auszublenden. Dies setzt allerdings eine etwas aktuellere Delphi-Version voraus, denn o.g. Benachrichtigung und die zu benutzende Reaktion darauf sind z.B. in Delphi 5 noch nicht bekannt. Das macht aber nichts, da wir uns die entsprechenden Daten aus dem aktuellen PSDK holen können:

   ```
   const
     TBN_INITCUSTOMIZE    = TBN_FIRST - 23;
     TBNRF_HIDEHELP       = $00000001;
   ```

   Nach der Definition im PSDK zu urteilen, sollte dafür aber mindestens der IE5 installiert sein. Um also den Hilfe-Button auszublenden, beantworten Sie die Benachrichtigung mit TBNRF_HIDEHELP:

   ```
   WM_NOTIFY:
     case PNMToolBar(lp)^.hdr.code of
       TBN_INITCUSTOMIZE:
         Result := TBNRF_HIDEHELP;
     end;
   ```

3. Es folgt eine TBN_QUERYINSERT-Benachrichtigung für jeden Button der Toolbar. Wenn Sie diese mit **false** (Null) beantworten, verhindern Sie, dass ein Button wieder in die Toolbar eingefügt werden kann, nachdem er entfernt wurde. Das gleiche gilt mit umgekehrten Vorzeichen für die Benachrichtigung TBN_QUERYDELETE: hier verhindern Sie mit **false**, dass der Button aus der Toolbar entfernt werden kann.

   Um z.B. den ersten Button so zu gestalten, dass er weder entfernt noch wieder eingefügt werden kann, müssten Sie beide Benachrichtigungen speziell für diesen Button mit

   ```
   if(PNMToolbar(lp)^.iItem = 0) then Result := LRESULT(false)
     else Result := LRESULT(true);
   ```

   beantworten. Problematisch wäre, wenn Sie nur TBN_QUERYINSERT auf diese Weise beantworten. Dann könnte der Button zwar aus der Toolbar entfernt, aber nicht mehr eingefügt werden.

   Der Dialog wird übrigens nicht angezeigt, wenn Sie alle TBN_QUERYINSERT-Benachrichtigungen mit **false** beantworten. Daher auch das anfängliche "Flackern".
4. Als nächstes steht eine TBN_GETBUTTONINFO-Benachrichtigung an; wieder für jeden Button. Sie müssen hier nichts weiter tun, als den jeweiligen Button aus Ihrer globalen Variable an das NMTOOLBAR-Record zu übertragen, das im lParam steckt:

   ```
   WM_NOTIFY:
     case PNMToolBar(lp)^.hdr.code of
       TBN_GETBUTTONINFO:
         begin
           nItem := PNMToolBar(lp)^.iItem;

           if(nItem < length(tbButtons)) then begin
             PNMToolBar(lp)^.tbButton := tbButtons[nItem];
   ```

   Wenn Ihre Buttons keinen eigenen Text benutzen (also nur Symbole darstellen), dann sollten Sie erklärenden Text zuweisen, weil Sie sonst auch nur die Symbole im Dialog sehen:

```
            lstrcpy(PNMToolBar(lp)^.pszText,
              pchar('Toolbutton ' + inttostr(nItem+1)));
          end;
```

Solange Buttons vorhanden sind, solange sollten Sie auf die Benachrichtigung mit **true** antworten. Sind keine Buttons mehr da, senden Sie **false**:

```
          Result := LRESULT(not(nItem=length(tbButtons)));
        end;
      end;
```

Damit sollten Sie die Dialogbox jetzt sehen.

3.4.7.3. Aktionismus

Wenn die Box sichtbar ist und vom Anwender benutzt wird, können Sie auf ein paar Benachrichtigungen reagieren, die alle als Teil von "`WM_NOTIFY`" gesendet werden.

3.4.7.4. Hilfe

Klickt der Anwender z.B. auf den Hilfe-Button (wenn Sie ihn nicht ausgeblendet haben), können Sie eine dazu passende Hilfeseite aufrufen:

```
TBN_CUSTHELP:
  MessageBox(wnd,'Hier könnte Ihre Hilfe erscheinen!','Toolbar-Demo',MB_OK or
MB_ICONINFORMATION);
```

Im Beispielprogramm wird allerdings nur diese Dialogbox angezeigt. Nun ja ... ;o)
Wenn Sie mehrere Toolbars benutzen und demzufolge auch verschiedene Hilfeseiten anzeigen lassen wollen, müssen Sie über "`PNMToolBar(lp)^.hdr.hwndFrom`" prüfen, welche Toolbar für diese Benachrichtigung verantwortlich ist.

3.4.7.5. Einstellungen wiederherstellen

Klickt der Anwender auf den Button "Zurücksetzen", dann sollten Sie die Originaleinstellungen der Toolbar wiederherstellen. Dazu entfernen Sie zuerst alle vorhandenen Buttons und weisen die Originalbuttons dann wieder neu zu:

```
TBN_RESET:
  begin
    nItem := SendMessage(PNMToolbar(lp)^.hdr.hwndFrom,TB_BUTTONCOUNT,0,0);

    // vorhandene Buttons entfernen
    for i := nItem - 1 downto 0 do
      SendMessage(PNMToolbar(lp)^.hdr.hwndFrom,TB_DELETEBUTTON,i,0);

    // Originalbuttons zuweisen
    SendMessage(PNMToolbar(lp)^.hdr.hwndFrom,TB_ADDBUTTONS,
      length(tbButtons),LPARAM(@tbButtons));
  end;
```

3.4.7.6. Änderungen übernehmen

Wenn der Anwender einen Button hinzugefügt oder entfernt hat oder anderweitig die Toolbar verändert hat, wird die Benachrichtigung `TBN_TOOLBARCHANGE` gesendet. Diese sollten Sie verwenden, um die Toolbar neu auszurichten; sprich: die Größe ggf. anpassen

```
TBN_TOOLBARCHANGE:
  SendMessage(PNMToolbar(lp)^.hdr.hwndFrom,TB_AUTOSIZE,0,0);
```

3.4.7.7. Einstellungen speichern und laden

All das würde nun aber wenig sinnvoll sein, wenn es keine Möglichkeit gäbe, diese Änderungen zu speichern und bei jedem Start zu laden. Diese Möglichkeit heißt "`TB_SAVERESTORE`" und wird an die Toolbar gesendet.
Das Schöne ist, dass wir uns um nichts weiter kümmern müssen. Die Toolbar nimmt uns die ganze Arbeit ab.

So wird über den `wParam` der o.g. Nachricht bestimmt, ob die Einstellungen gespeichert (**true**) oder geladen (**false**) werden sollen. Der `lParam` ist ein Zeiger auf ein Record vom Typ `TTBSaveParams`, das die notwendigen Daten enthält. Weil die Informationen in der Registry gespeichert werden, ist im Record lediglich der Hauptschlüssel, der Schlüsselname (Ihrer Software, z.B.) und der Name des Wertes anzugeben. Letzterer enthält die eigentlichen Toolbardaten; d.h.: wenn Sie mehrere Toolbars verwenden, müssen Sie die Nachricht entsprechend oft senden und immer den passenden Wertenamen angeben. Da unser nur eine Toolbar hat, brauchen wir auch nur einen Namen:

```
sp.hkr           := HKEY_CURRENT_USER;
sp.pszSubKey     := 'Software\Win32-API-Tutorials\Toolbar-Demo';
sp.pszValueName  := 'ToolbarSettings';
```

Im Hinblick auf NT-Betriebssysteme und evtl. eingeschränkte Benutzerrechte empfehle ich grundsätzlich den Schlüssel `HKEY_CURRENT_USER` zum Speichern. Wenn Ihr Programm seine Einstellungen sowieso in der Registry ablegt, dann sollten Sie die Toolbar-Einstellungen im selben Schlüssel unterbringen.

Zum Speichern benutzen wir nun, wie gesagt!, die Nachricht "`TB_SAVERESTORE`" mit dem `wParam` **true**:

```
SendMessage(hToolbar,TB_SAVERESTORE,WPARAM(true),LPARAM(@sp));
```

Wollen Sie die Einstellungen laden, geben Sie stattdessen **false** an.

3.5. Das IP-Adress-Eingabefeld

3.5.1. Das IP-Control erzeugen

Das IP-Eingabefeld ist Ihnen bestimmt aus einigen Dialogen des Betriebssystems bekannt. Am besten lässt es sich noch mit einem normalen Eingabefeld vergleichen. Es ist nur in vier Felder aufgeteilt, die die gezielte Bearbeitung einer IP-Adresse ermöglichen:

Wie jedes "Common Control" muss auch dieses Eingabefeld initialisiert werden. Allerdings benutzen wir hier die Funktion "`InitCommonControlsEx`", weil wir eine spezielle Klasse angeben müssen:

```
var
  icc : TInitCommonControlsEx = (
    dwSize : sizeof(TInitCommonControlsEx);
    dwICC  : ICC_INTERNET_CLASSES;
  );

begin
  InitCommonControlsEx(icc);

  { ... }
end.
```

Danach kann das IP-Eingabefeld wie jedes normale Fenster erzeugt werden. Zu beachten ist die Konstante `WC_IPADDRESS`, die man anstelle des eigentlichen Klassennamens `SysIPAddress32` verwenden kann:

```
hIpAddr := CreateWindowEx(WS_EX_CLIENTEDGE,WC_IPADDRESS,nil,
  WS_VISIBLE or WS_CHILD,10,20,120,21,wnd,IDC_IPCTRL,
  hInstance,nil);
```

3.5.2. IP-Adressen eintragen und auslesen

Zum Eintragen einer IP-Adresse stehen uns die Nachricht "`IPM_SETADDRESS`" und die Funktion "`MAKEIPADDRESS`", zur Verfügung. Normalerweise handelt es sich bei einer IP-Adresse nämlich um einen 32-Bit-Wert, der aber (zur besseren Lesbarkeit) in vier Felder zu je 8 Bit aufgeteilt ist, die durch Punkte voneinander getrennt sind ("doted quad notation"). So entspricht z.B. der Wert 2130706433 (0x7f000001) der IP-Adresse "127.0.0.1". Und wenn man nicht umständlich den `DWORD`-Wert einer Adresse ausrechnen möchte, kann man die Hilfe von "`MAKEIPADDRESS`" direkt die einzelnen Bytes angeben:

```
if(hIpAddr <> 0) then
  SendMessage(hIpAddr,IPM_SETADDRESS,0,MAKEIPADDRESS(127,0,0,1));
```

IPM_SETADDRESS-Definition

```
IPM_SETADDRESS
    wParam = (WPARAM) 0                 // nicht genutzt
    lParam = (LPARAM) (DWORD) dwAddr    // IP-Adresse als DWORD
```

MAKEIPADDRESS-Definition

```
LPARAM MAKEIPADDRESS(
    BYTE b0,
    BYTE b1,
    BYTE b2,
    BYTE b3
);
```

Das Auslesen der aktuellen IP-Adresse aus dem Control führen wir mit Hilfe der Nachricht "IPM_GETADDRESS" durch, die als LPARAM-Wert einen Zeiger auf eine DWORD-Variable erwartet, was in unserem Beispiel so aussieht:

```
var
  curIp : dword;

{ ... }
```

IPM_GETADDRESS-Definition

```
IPM_GETADDRESS
    wParam = (WPARAM) 0                 // nicht genutzt
    lParam = (LPARAM) (LPDWORD) dwAddr // Zeiger auf DWORD-Variable
```

Damit wir in unseren Programmen ebenfalls die übliche Schreibweise benutzen können, enthält die Unit "CommCtrl.pas" vier Funktionen, mit denen wir die einzelnen Bytes aus unserer Variablen extrahieren können:

```
FIRST_IPADDRESS (LPARAM lParam)
SECOND_IPADDRESS(LPARAM lParam)
THIRD_IPADDRESS (LPARAM lParam)
FOURTH_IPADDRESS(LPARAM lParam)
```

Alle vier Funktionen haben gemeinsam, dass sie als Parameter den DWORD-Wert der IP-Adresse erwarten. Jede liefert dann allerdings ein anderes Byte der Adresse zurück. Würde man die Beispieladresse "127.0.0.1" z.B. von "FIRST_IPADDRESS" bearbeiten lassen, wäre das Ergebnis "127". Bei "SECOND_IPADDRESS" dagegen wäre das Ergebnis Null, usw.
Das Beispielprogramm nutzt alle vier Funktionen, um die IP-Adresse formatiert an ein Label weiterzugeben:

```
IpStr := Format('%d.%d.%d.%d',
  [FIRST_IPADDRESS(curIp),SECOND_IPADDRESS(curIp),
   THIRD_IPADDRESS(curIp),FOURTH_IPADDRESS(curIp)]);

SendMessage(hLabel,WM_SETTEXT,0,LPARAM(@IpStr[1]));
```

3.5.3. Das Eingabefeld leeren

Um das komplette IP-Eingabefeld zu leeren, verwenden wir die Nachricht "IPM_CLEARADDRESS", die keine besonderen Parameter erwartet:

```
SendMessage(hIpAddr,IPM_CLEARADDRESS,0,0);
```

3.5.4. Einzelne Felder fokussieren

Bei einem IP-Eingabefeld hat man die Möglichkeit, gezielt ein Feld auszuwählen und so beispielsweise eine Änderung durch den Benutzer zu verlangen. Das Beispielprogramm führt diese Fokussierung mit Hilfe eines Buttons vor, der nacheinander die einzelnen Felder ansteuert:

```
WM_COMMAND:
  case HIWORD(wp) of
    BN_CLICKED:
      case LOWORD(wp) of
        IDC_FOCUS:
          begin
            iFocus := (iFocus + 1) mod 4;
            SendMessage(hIpAddr,IPM_SETFOCUS,iFocus,0);
          end;
      end;
  end;
```

Benutzt wird hier eine `integer`-Variable, die bei der Initialisierung den Wert -1 erhalten hat. Das hat damit zu tun, dass die Zählweise der IP-Bytefelder mit Null beginnt. Durch den Buttonklick wird die Variable um Eins erhöht und entspricht somit dem tatsächlichen Feldzähler, der mit der Nachricht "`IPM_SETFOCUS`" an das IP-Eingabefeld weitergeleitet wird.

IPM_SETFOCUS-Definition zeigen

```
IPM_SETFOCUS
    wParam = (WPARAM) (int) nField; // Feld, das den Fokus erhalten soll
    lParam = (LPARAM) 0             // nicht genutzt
```

Auch wenn damit ein wenig auf das nächste Thema vorgegriffen wird - Bei jedem Wechsel von einem Bytefeld zum anderen, bzw. beim Ändern des Wertes in einem Bytefeld, wird die Variable `iFocus` entsprechend angepasst, so dass sie immer das aktuell gewählte Bytefeld enthält.

3.5.5. Auf Änderungen reagieren

Das IP-Eingabefeld sendet die Nachricht "`IPN_FIELDCHANGED`" als Teil von "`WM_NOTIFY`", sobald sich irgendetwas ändert. Das betrifft allerdings nicht nur die Änderung von Werten der einzelnen Bytefelder. Die Nachricht wird auch gesendet, wenn man sich innerhalb des Controls mit der Maus oder den Pfeiltasten von einem Bytefeld zum nächsten bewegt.

Das Beispielprogramm verdeutlicht dieses Prinzip, in dem es bei jedem Auftreten der Nachricht den Wert des geänderten Feldes heraussucht und im Label anzeigt:

```
WM_NOTIFY:
  if(PNMIpAddress(lp)^.hdr.Code = IPN_FIELDCHANGED) then
    begin
      IpStr := Format('Feld "%d" enthält Wert "%d"',
        [POINTER(PNMIpAddress(lp)^.iField + 1),
         POINTER(PNMIpAddress(lp)^.iValue)]);
      SendMessage(hLabel,WM_SETTEXT,0,LPARAM(@IpStr[1]));
    end;
```

Außerdem besitzt das IP-Feld eine typische Eigenschaft der bekannten Eingabefelder: wird etwas geändert, wird "`EN_CHANGE`" ausgelöst. Das heißt: Sie können bereits während der Eingabe reagieren. Das kann z.B. hilfreich sein, wenn Sie bestimmte Elemente in Abhängigkeit von einer korrekten Ein- oder Angabe aktivieren oder deaktivieren möchten.

```
WM_COMMAND:
  if(HIWORD(wp) = EN_CHANGE) and (LOWORD(wp) = IDC_IPCTRL) then
    begin
      { ... }
    end;
```

3.6. Die Trackbar

3.6.1. Das Trackbar-Control erzeugen

Erzeugt wird die Trackbar wie jedes andere Fenster mit der Funktion "CreateWindowEx", wobei die Fensterklasse msctls_trackbar32 heißt (oder alternativ TRACKBAR_CLASS):

```
hredTB := CreateWindowEx(0, 'msctls_trackbar32', '', WS_VISIBLE or WS_CHILD or
  WS_TABSTOP or TBS_TOP or TBS_AUTOTICKS or TBS_TOOLTIPS,
  10, 15, 275, 35, hWnd, 0, hInstance, nil);
```

InitCommonControlsEx

Wenn Sie den Befehl "InitCommonControlsEx" verwenden, müssen Sie für die dwICC-Membervariable die Klasse ICC_BAR_CLASSES benutzen.

3.6.2. Fensterstile der Trackbar

Ich stelle hier gleich noch die Fensterstile vor, die ich auch in meinem Programm verwendet habe. Weitere lassen sich im MSDN oder im PSDK nachschlagen:

Wert	Beschreibung
TBS_TOP	Die Teilstriche werden oberhalb angezeigt.
TBS_BOTH	Die Teilstriche werden oberhalb und unterhalb angezeigt.
TBS_AUTOPTICKS	Der Abstand der Teilstriche wird automatisch gesetzt. Der Abstand kann mit der Nachricht "TBM_SETTICFREQ" bestimmt werden.
TBS_TOOLTIPS	Beim Bewegen des Sliders wird der aktuelle Wert als Tooltip angezeigt.

3.6.3. Trackbar-Nachrichten

Mit den im Beispielprogramm verwendeten folgenden Nachrichten wird das Verhalten der Trackbar noch weiter definiert bzw. beeinflusst:

Wert	Beschreibung
TBM_SETRANGE	Mit dieser Nachricht wird der Wertebereich der Trackbar bestimmt, sprich: der minimale und maximale Wert, der eingestellt werden kann.
TBM_SETTICFREQ	Verwendet man beim Erstellen der Trackbar den Fensterstil "TBS_AUTOTICKS", kann man mit dieser Nachricht den Abstand der Teilstriche festlegen.
TBM_SETLINESIZE	Der Slider der Trackbar kann auch mit den Pfeiltasten bewegt werden. Mit dieser Nachricht legt man fest, um wie viele Einheiten der Slider bewegt werden soll.
TBM_SETPAGESIZE	Entsprechende Nachricht für die "Bild auf"- und "Bild ab"-Tasten.

Auch hier habe ich nur die wichtigsten aufgeführt. Weitere Nachrichten sind im MSDN oder PSDK zu finden.

Die Nachricht "TBM_SETRANGE" braucht wohl noch eine Erläuterung wegen des zweiten Parameters. Schauen wir uns das mal im Code an:

```
SendMessage(hredTB, TBM_SETRANGE, Integer(TRUE), MAKELONG(0,255));
```

In C ist "MAKELONG" ein Makro, das aus zwei Integerwerten einen 32-Bit-Wert generiert. Bei lParam handelt es sich nämlich um einen 32-Bit-Wert, während wParam früher auf 16-Bit-Systemen ein 16-Bit-Wert war. Heutzutage handelt es sich aber auch bei wParam um einen 32-Bit-Wert.
Aus dieser Tatsache lässt sich auch das "w" vor wParam und das "l" vor lParam erklären: "w" stand wohl für WORD und "l" für LONGWORD.

Das heißt für uns konkret: das niederwertige Wort legt die untere Grenze des Wertebereichs fest und das höherwertige Wort die obere Grenze.

3.6.4. Funktionsweise des Programms

Hier nun noch ein paar Anmerkungen zur Funktionsweise des Programms: Verändert man die Position des Schiebereglers einer Trackbar, wird von Windows die Fensternachricht "WM_HSCROLL" an das Fenster gesendet.

Hinweis

Wenn Sie einen vertikalen Regler erzeugen (dazu benutzen Sie das Stilattribut TBS_VERT), dann müssen Sie stattdessen die Nachricht "WM_VSCROLL" bearbeiten. "WM_HSCROLL" wird nur bei den normalen, horizontalen Reglern gesendet. Abgesehen davon sind aber alle nachfolgenden Informationen identisch. Zur Anschauung finden Sie im Ordner mit den Beispielprogrammen eine zweite Variante namens "Vertical.dpr".

Im niederwertigen Wort von wParam steckt der Benachrichtigungscode, der Auskunft gibt, wie der Benutzer die Position des Schiebereglers verändert hat (ob mit der Maus, den Pfeiltasten, oder ob er mit der Maus links oder rechts neben den Schieberegler geklickt hat ... wie auch immer ...). Auf alle Möglichkeiten habe ich im Programm reagiert.

```
WM_HSCROLL:
  begin
    case LoWord(wParam) of
      TB_THUMBTRACK,  // ziehen des "Sliders"
      TB_TOP,         // Pos1
      TB_BOTTOM,      // Ende
      TB_LINEUP,      // Pfeiltasten links/rechts
      TB_LINEDOWN,    // Pfeiltasten oben/unten
      TB_PAGEDOWN,    // Bild runter & in die Leiste geklickt
      TB_PAGEUP:      // Bild auf & in die Leiste geklickt
        begin
          // entsprechende Aktion auslösen
        end;
    end;
  end;
```

Bleibt noch das höherwertige Wort von wParam: hier steht die aktuelle Position des Schiebereglers.

Benötigt man eine Identifikation, welcher Schieberegler die Nachricht gesendet hat, so braucht man nur den lParam auszulesen, in dem das Handle der entsprechenden Trackbar übergeben wird. Ich habe darauf verzichtet und es mir einfach gemacht:

Muss ein Fenster neu gezeichnet werden (weil der Fensterinhalt ungültig geworden ist, weil es verdeckt oder verschoben wurde ...), bekommt es eine "WM_PAINT"-Nachricht. Hier kann man entsprechende Zeichenoperationen durchführen. Ich lese z.B. an dieser Stelle im Programm die Positionen der Schieberegler aus, erzeuge mit "CreateSolidBrush" einen *Pinsel* mit Hilfe dieser Werte, und fülle damit ein, mit einem TRect-Record festgelegtes Rechteck:

```
WM_PAINT:
  begin
    dc    := BeginPaint(hWnd, ps);
    red   := Sendmessage(hredTB, TBM_GETPOS, 0, 0);
    green := Sendmessage(hgreenTB, TBM_GETPOS, 0, 0);
    blue  := Sendmessage(hblueTB, TBM_GETPOS, 0, 0);

    rect.Top    := 160;
    rect.Left   := 0;
    rect.Bottom := WindowHeight;
    rect.Right  := WindowWidth;
    brush       := CreateSolidBrush(RGB(red, green, blue));

    FillRect(dc, rect, brush);
  end;
```

Des Weiteren gebe ich die aktuellen Farbwerte als invertierten Text aus. Invertiert deshalb, damit man sie auch noch lesen kann. Dazu setze ich den Hintergrundfarbmodus mit "`SetBkMode`" auf transparent, bastle mir den jeweiligen String zusammen und gebe ihn mit "`TextOut`" direkt auf dem Fenster aus.

Für diese Aktionen ist ein "`DeviceContext`", (ein Handle auf die Zeichenfläche) nötig, den man mit "`BeginPaint`" bekommt. Der Parameter, ein `TPaintStruct`-Record, interessiert uns dabei aber nicht - wir brauchen nur den Rückgabewert: unseren "`DeviceContext`".

Wichtig ist, dass wir unsere Zeichenaktionen mit "`EndPaint`" wieder abschließen und Windows so mitteilen, das wir mit dem Aktualisieren unseres Fensters fertig sind. Tun wir dies nicht, bekäme unser Programm laufend weiter "`WM_PAINT`"-Nachrichten. Schlimmstenfalls reagiert unser Programm dann gar nicht mehr.

Und da ich nicht warten will, bis das Fenster neu gezeichnet werden muss, erzwinge ich das Neuzeichen mit Hilfe von "`InvalidateRect`". Aber das passiert in der o.g. Bearbeitung der "`WM_HSCROLL`"-Nachricht:

```
InvalidateRect(hWnd, nil, TRUE);
```

Hinweis

Mit Zeichen- und Textausgabeoperationen werde ich mich in einem anderen Tutorial noch mal ausführlicher befassen.

3.7. Das List-View-Control

3.7.1. Das List-View-Control erzeugen

Aus der VCL-Programmierung und selbst aus dem ganz normalen Windows-Alltag dürfte Ihnen die List-View bekannt sein. Wer den Explorer benutzt, kennt sie; diverse andere Programme verwenden sie auch. Und auch uns bereitet es keine Schwierigkeiten, ein solches Control in unseren nonVCL-Programmen einzusetzen.

Es gibt mehrere Darstellungsarten, von denen Sie eine bei der Erzeugung des Fensters (nichts anderes ist unser Control ja) angeben müssen:

Wert	Bedeutung
LVS_REPORT	Detailansicht inkl. Spaltenköpfe, die zur Sortierung benutzt werden können.
LVS_LIST	Listenansicht
LVS_ICON	Alle Einträge werden mit großen Symbolen dargestellt.
LVS_SMALLICON	Alle Einträge werden mit kleinen Symbolen dargestellt.

Die Fensterklasse unseres Controls ist `SysListView32` (bzw. die Konstante `WC_LISTVIEW`), und der Aufruf sieht dann so aus:

```
hLV      := CreateWindowEx(WS_EX_CLIENTEDGE,WC_LISTVIEW,nil,
  WS_VISIBLE or WS_CHILD or LVS_REPORT or LVS_EDITLABELS or
  LVS_SHOWSELALWAYS or LVS_SHAREIMAGELISTS,0,0,100,100,wnd,
  IDC_LV,hInstance,nil);
```

InitCommonControlsEx

Wenn Sie den Befehl "`InitCommonControlsEx`" verwenden, müssen Sie für die `dwICC`-Membervariable die Klasse `ICC_LISTVIEW_CLASSES` benutzen.

3.7.2. Spezielle Fensterstile

Eine List-View kann man mit besonderen Effekten ausstatten, als da wären:

- Selektieren einer ganzen Zeile
- Hovering
- Gridlinien
- Symbole für Untereinträge

usw. Im PSDK finden Sie unter den Stichworten "`LVM_SETEXTENDEDLISTVIEWSTYLE`" und "Extended List-View Styles" weitere Möglichkeiten, wobei aber zu beachten ist, dass einige davon nur unter bestimmten Versionen der "common controls" verfügbar sind.

Gesetzt werden solche erweiterten Stile aber in jedem Fall wie folgt:

```
SendMessage(hLV,LVM_SETEXTENDEDLISTVIEWSTYLE,0,
  LVS_EX_HEADERDRAGDROP or LVS_EX_FULLROWSELECT);
```

3.7.3. Spalten erzeugen

Bis jetzt wird man nicht viel Spaß an der List-View haben, da die Spalten noch erzeugt werden müssen. Dies gilt übrigens auch dann, wenn man das Control über eine Dialog-Ressource erzeugt. Wir benötigen dazu ein Record vom Typ `TLVColumn`, von dem uns aber nicht alle Parameter zwangsläufig interessieren müssen. Zuerst legen wir in der Maske die gültigen Membervariablen fest:

```
lvc.mask    := LVCF_TEXT or LVCF_WIDTH;
```

Uns interessieren hier also Text und Breite. Den Spaltentext geben wir wie folgt an

```
lvc.pszText := 'Datei';
```

Fehlt noch die Spaltenbreite

```
lvc.cx      := 200;
```

Dann können wir die Spalte nun erzeugen:

```
SendMessage(hLV,LVM_INSERTCOLUMN,0,LPARAM(@lvc));
```

Für die zweite Spalte wollen wir die Textausrichtung (rechtsbündig) ändern, was folgende Auswirkung auf die Maske hat:

```
lvc.mask    := lvc.mask or LVCF_FMT;
```

Also geben wir auch gleich den Wunsch nach rechtsbündigem Text an

```
lvc.fmt     := LVCFMT_RIGHT;
```

Text und Spaltenbreite werden wie bekannt übergeben:

```
lvc.pszText := 'Größe';
lvc.cx      := 150;
```

Und damit können wir die zweite Spalte erzeugen:

```
SendMessage(hLV,LVM_INSERTCOLUMN,1,LPARAM(@lvc));
```

Die dritte Spalte wird später den Dateityp anzeigen. Da es beim Erzeugen der Spalte nun keine großen Neuheiten mehr gibt, verweise ich Sie an dieser Stelle auf das >Beispielprogramm, das den vollständigen Code in der Prozedur "`MakeColumns`" enthält.

TLVColumn-Definition

```
typedef struct _LVCOLUMN {
    UINT mask;             // Eigenschaften (hier Spalte, & Beschriftung)
    int fmt;               // Ausrichtung der Spalte
    int cx;                // Spaltenbreite
    LPTSTR pszText;        // Spaltentext
    int cchTextMax;
    int iSubItem;
#if (_WIN32_IE >= 0x0300)
    int iImage;
    int iOrder;
#endif
}
```

LVM_INSERTCOLUMN-Definition

```
LVM_INSERTCOLUMN
    wParam = (WPARAM) (int) iCol;                    // Spaltenindex
    lParam = (LPARAM) (const LPLVCOLUMN) pcol;       // Zeiger auf das
"TVColumn"-Record
```

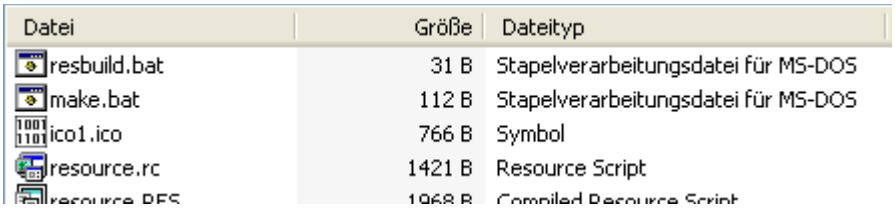
Alternative

Anstelle der Nachricht "`LVM_INSERTCOLUMN`" können Sie auch die Funktion " `ListView_InsertColumn`" benutzen:

```
ListView_InsertColumn(hLV,1,@lvc);
```

Hinweis für Windows XP

Sicher kennen Sie den optischen Effekt im Explorer von Windows XP, der die gerade aktuelle Spalte grau hinterlegt. Dies ist eine Funktion, die Ihnen das Betriebssystem zur Verfügung stellt; eine aufwändige Programmierung entfällt:



Das setzt natürlich voraus, dass Windows XP läuft und dass Ihre Delphi-Version aktuell genug ist und die benötigte Funktion, "`ListView_SetSelectedColumn`", kennt. Außerdem benötigt das Programm eine Manifestdatei, damit die Common Controls 6.0 geladen werden können. Diese Datei kennen Sie evtl. schon, wenn Sie das jeweils eingestellte XP-Thema auch für Ihr Programm benutzen wollen. Ob sie dem Programm einfach nur beiliegt ("`Dateiname.exe.manifest`") oder in den Ressourcen eingebunden ist, spielt dabei keine Rolle. Wichtig ist, dass das Programm darauf zugreifen kann; ansonsten sehen Sie obigen Effekt nicht!

Falls Sie eine ältere Delphi-Version haben, können Sie die Unit "`CommCtrl_Fragment.pas`" benutzen, die diesem Tutorial beiliegt. Die Unit wurde erfolgreich mit Delphi 5 getestet und enthält ein paar der XP-Aktualisierungen; u.a. auch die o.g. Funktion.

Das Beispielprogramm geht nun wie folgt vor: Beim Start wird die erste Spalte markiert, wobei Sie der Funktion einfach den Index der Spalte (null-basierend) übergeben:

```
ListView_SetSelectedColumn(hLV,0);
```

Die gleiche Funktion wird auch bei der Sortierung verwendet. Klicken Sie auf einen Spaltenkopf, werden die Einträge nicht nur sortiert - unter Windows XP wird dann auch die entsprechende Spalte markiert.

Mit der Nachricht "`LVM_GESELECTEDCOLUMN`" bzw. dem Makro "`ListView_GetSelectedColumn`" können Sie übrigens die aktuell markierte Spalte ermitteln:

```
iCol := ListView_GetSelectedColumn(hLV);
```

3.7.4. Die List-View mit Inhalt füllen

Für das Füllen benötigen wir ein weiteres Record, `TLVItem`. Als Beispiel soll ein Auszug aus der entsprechenden Prozedur unseres Programms> dienen. Man erzeugt eine Maske, die den Typ (Text) bestimmt

```
lvi.mask     := LVIF_TEXT;
```

Dann legt man den Item- und SubItem-Index fest

```
lvi.iItem    := Loop;
lvi.iSubItem := 0;
```

weist den Text zu (in dem Fall: der Name der gefundenen Datei)

```
lvi.pszText  := finddata.cFileName;
```

Damit können wir den Eintrag mit der Nachricht "LVM_INSERTITEM" in die Liste einfügen, wobei der lParam-Wert der Zeiger auf das eben gefüllte Record ist:

```
SendMessage(hLV,LVM_INSERTITEM,0,Integer(@lvi));
```

Alternative
Alternativ dazu können Sie auch "ListView_InsertItem" verwenden, das nur eine Kapselung der Nachricht ist:

```
ListView_InsertItem(hLV,lvi);
```

Für unseren zweiten Eintrag lesen wir zunächst die Dateigröße in einen Textpuffer. Weil es sich nun um den Eintrag in der zweiten Spalte handelt, müssen wir die iSubItem-Membervariable des Records entsprechend ändern:

```
lvi.iSubItem := 1;
```

Wir weisen den Textpuffer mit der Dateigröße zu

```
lvi.pszText  := buf;
```

und benutzen diesmal die Nachricht "LVM_SETITEM" bzw. das Makro "ListView_SetItem"

```
ListView_SetItem(hLV,lvi);
```

Der Unterschied ist, dass "LVM_INSERTITEM" ein neues Item einfügt, während wir mit "LVM_SETITEM" die Möglichkeit haben, weitere Angaben zum aktuellen Item (wie eben Einträge in anderen Spalten) zu treffen.

TLVItem-Definition

```
typedef struct _LVITEM {
    UINT mask;              // gültige Eigenschaften
    int iItem;              // Item-Index
    int iSubItem;
    UINT state;
    UINT stateMask;
    LPTSTR pszText;         // Item-Text
    int cchTextMax;
    int iImage;             // Image-Index
    LPARAM lParam;
#if (_WIN32_IE >= 0x0300)
    int iIndent;
#endif
#if (_WIN32_IE >= 0x560)
    int iGroupId;
    UINT cColumns;
    PUINT puColumns;
#endif
}
```

LVM_INSERTITEM-Definition

```
LVM_INSERTITEM
    wParam = 0;
    lParam = (LPARAM) (const LPLVITEM) pitem;
```

3.7.5. Den Dateityp bestimmen

Eine kleine Besonderheit ist die Anzeige des Dateityps. Diese Namen werden vom System zur Verfügung gestellt, und wir kommen mit Hilfe der Funktion "SHGetFileInfo" an sie heran. Die Funktion benötigt neben dem Dateinamen eine Variable vom Typ "TSHFileInfo". Da wir an der Typenbezeichnung interessiert sind, benutzen wir das Flag SHGFI_TYPENAME, so dass der Aufruf wie folgt aussieht:

```
SHGetFileInfo(finddata.cFilename,0,fi,sizeof(TSHFileInfo),SHGFI_TYPENAME);
```

Nach dem Aufruf enthält die szTypeName-Membervariable die Bezeichnung. Der Rest entspricht der o.g. Vorgehensweise: Sie wählen den Index des Sub-Items, damit die passende Spalte gefüllt wird, dann Sie weisen den ausgelesenen Typennamen zu und fügen das neue Item in die List-View ein.

```
lvi.mask     := LVIF_TEXT;
lvi.iSubItem := 2;
lvi.pszText  := fi.szTypeName;
ListView_SetItem(hLV,lvi);
```

3.7.6. Bildchen wollen Sie auch noch ...?

Na gut. Was wäre so eine List-View auch ohne die typischen Symbole? In diesem Kapitel wollen wir uns zuerst anschauen, wie wir eigene Grafiken verwenden können. Dazu benötigen wir eine so genannte ImageList. Diese wird mit Symbolen gefüllt und der List-View zugewiesen.

Sollte es notwendig sein, dann müssen wir je eine ImageList für die großen und eine für die kleinen Symbole erstellen. Wir schauen uns das anhand der kleinen Symbole einmal an: Mit "ImageList_Create" erzeugen wir die Image-Liste. Als Rückgabewert erhalten wir ein Handle, das wir nachher noch für die Zuweisung an das List-View-Control brauchen:

```
hImgSm  := ImageList_Create(16,16,ILC_COLOR,0,1);
```

ImageList_Create-Definition>

```
HIMAGELIST ImageList_Create(
    int cx,             // Breite, muss nicht der tatsächlichen entsprechen
    int cy,             // Höhe, muss nicht der tatsächlichen entsprechen
    UINT flags,         // Flags, hauptsächlich für die Farben
    int cInitial,       // Anzahl der Images beim Erzeugen, kann Null sein
    int cGrow           // Anzahl der Images, um die die Liste erhöht werden
                        // kann, wenn das System neue Images hinzufügen muss
   );
```

Als nächstes füllen wir die Liste mit einem Symbol aus der Ressourcendatei:

```
ImageList_AddIcon(hImgSm,LoadIcon(hInstance,MAKEINTRESOURCE(1)));
```

ImageList_AddIcon-Definition:

```
int ImageList_AddIcon(
    HIMAGELIST himl,
    HICON hicon
   );
```

Auf die gleiche Weise ließen sich natürlich noch weitere Symbole an die ImageList anhängen. Damit die List-View die Symbole aber auch nutzen kann, weisen wir die Liste mit der Nachricht "LVM_SETIMAGELIST" zu:

```
SendMessage(hLV,LVM_SETIMAGELIST,LVSIL_SMALL,hImgSm);
```

Der zweite Parameter ist das Handle unserer ImageList, und der erste Parameter beschreibt den Typ der Liste:

Wert	Bedeutung
LVSIL_NORMAL	Imagelist mit großen Symbolen
LVSIL_SMALL	Imagelist mit kleinen Symbolen
LVSIL_STATE	Imagelist mit Statussymbolen

LVM_SETIMAGELIST-Definition

```
LVM_SETIMAGELIST
    wParam = (WPARAM) (int) iImageList;
    lParam = (LPARAM) (HIMAGELIST) himl;
```

Alternative:
Diese Nachricht wird von der Funktion "ListView_SetImageList" gekapselt, die Sie ebenfalls verwenden können. Allerdings geben Sie das Handle der ImageList und den Typ in umgekehrter Reihenfolge an, z.B.:

```
ListView_SetImageList(hLV,hImgSm,LVSIL_SMALL);
```

Bei der Zuweisung der Liste für die großen Symbole ist ebenso vorzugehen. Bleibt nur noch die Frage: Wie kann ein Item in der List-View ein solches Symbol verwenden? Dazu möchte ich Sie an das letzte Kapitel erinnern, in dem es um das Füllen der List-View ging. Wenn Sie den ersten Eintrag (bezogen auf die Spalte mit dem Index Null) in die List-View eintragen wollen, ergänzen Sie bitte die Maske wie folgt:

```
lvi.mask      := LVIF_TEXT or LVIF_IMAGE;
```

Damit steht Ihnen die Membervariable iImage zur Verfügung, der Sie den Index des gewünschten Symbols übergeben. Wenn wir davon ausgehen, dass wir nur ein Symbol in der ImageList haben, lautet der Aufruf also ganz einfach nur:

```
lvi.iImage    := 0;
```

Dann wird der Eintrag wie im letzten Kapitel beschrieben mit "LVM_INSERTITEM" (bzw. "ListView_InsertItem") eingetragen und sollte das Symbol dann auch anzeigen.
Ich habe übrigens festgestellt, dass durch die Zuweisung einer ImageList automatisch das erste Symbol für alle Items benutzt wird, wenn Sie keine speziellen Angaben treffen. Wenn Ihre Einträge also alle das selbe (erste) Symbol der Liste anzeigen sollen, dann können Sie auf das Flag LVIF_IMAGE verzichten und die iImage-Membervariable ignorieren.

Bilder für Sub-Items

In der Detailansicht (LVS_REPORT-Stil) können Sie auch die Einträge in anderen Spalten mit Grafiken versehen. Dazu verwenden Sie, wie eben gezeigt, eine entsprechende Maske und übergeben den Indexwert des gewünschten Bildes, beispielsweise

```
lvi.mask      := LVIF_TEXT or LVIF_IMAGE;
lvi.iImage    := 1;
```

usw. Zusätzlich müssen Sie aber ein erweitertes Stilattribut setzen, sonst werden die Grafiken nicht angezeigt:

```
SendMessage(hLV,LVM_SETEXTENDEDLISTVIEWSTYLE,0,
  { ... } or LVS_EX_SUBITEMIMAGES);
```

Soll die List-View in dem Fall vor dem ersten Eintrag kein Symbol anzeigen, dann erinnern Sie sich bitte daran, was ich eben schrieb: Durch die Zuweisung einer ImageList wird automatisch das erste Symbol benutzt, selbst wenn Sie das nicht explizit angeben.
Sie können dieses Verhalten nur durch einen "Trick" umgehen, indem Sie nämlich das Flag LVIF_IMAGE für den ersten Eintrag benutzen, der iImage-Variablen aber den Wert -1 (I_IMAGENONE) übergeben:

```
lvi.mask      := LVIF_TEXT or LVIF_IMAGE;
lvi.iImage    := I_IMAGENONE;
```

Das hat allerdings den recht unschön wirkenden Nachteil, dass trotzdem links neben dem Eintrag der Platz für die Grafik frei ist. Das hängt aber mit der Zuweisung der Imageliste zusammen. Nur wenn Sie keine Liste verwenden, sind die Einträge direkt am linken Rand.

Abgesehen davon sieht das bestenfalls im Report-Modus gut aus. Sobald Sie eine andere Ansicht verwenden, wird standardmäßig wieder der erste Eintrag mit seinem (dann evtl. nicht vorhandenen) Symbol benutzt, was u.U. zu merkwürdigen Ergebnissen führt.

3.7.6.1. Die ImageList freigeben

Mit der Funktion "ImageList_Destroy" wird der Speicher für die Liste wieder freigegeben. Das also bitte nicht bei "WM_DESTROY" vergessen:

```
case uMsg of
  WM_DESTROY:
    ImageList_Destroy(hImgSm);
end;
```

ImageList_Destroy-Definition:

```
BOOL ImageList_Destroy(
    HIMAGELIST himl
   );
```

3.7.6.2. Symbole, die Zweite!

Eine noch elegantere Methode ist die, das System für die Symbole heranzuziehen. Da unser Beispielprogramm ja sowieso die Dateien des aktuellen Ordners einliest, können wir uns auch ebenso gut die entsprechenden Dateisymbole anzeigen lassen.

Weil wir aber zwei Imagelisten gleichzeitig benutzen wollen (für die kleinen und großen Symbole), ist es sehr wichtig, dass die List-View mit dem Stilattribut LVS_SHAREIMAGELISTS erzeugt wurde. Das hat damit zu tun, dass normalerweise bei der Zuweisung einer Imageliste eine evtl. bereits benutzte wieder freigegeben wird. Sie können also nur mit dem genannten Stilattribut mehr als eine Imageliste verwenden.
Das gilt natürlich auch für den Fall, dass Sie eigene Symbole benutzen wollen. Aber zumindest sind hier die Folgen sofort sichtbar, wenn Sie das Attribut vergessen: Ihre Symbole auf dem Desktop, im Explorer usw. verschwinden möglicherweise und sind erst nach einem Neustart wieder zu sehen.

Wenn Sie also überprüft haben, dass das o.g. Attribut benutzt wurde, dann können Sie die beiden Imagelisten laden. Wir benutzen hierfür die Funktion "SHGetFileInfo", die uns das Handle der jeweiligen Liste als Ergebnis liefert, wenn wir das Flag SHGFI_SYSICONINDEX benutzen.
Für die kleinen Symbole sieht der Aufruf so aus:

```
hImgSm  := HIMAGELIST(SHGetFileInfo('',0,fi,sizeof(fi),
  SHGFI_SYSICONINDEX or SHGFI_SMALLICON));
if(hImgSm <> 0) then
  ListView_SetImageList(hLV,hImgSm,LVSIL_SMALL);
```

Für die großen Symbole tauschen wir nur das Flag SHGFI_SMALLICON gegen SHGFI_ICON aus:

```
hImgBig := HIMAGELIST(SHGetFileInfo('',0,fi,sizeof(fi),
  SHGFI_SYSICONINDEX or SHGFI_ICON));
```

Und natürlich müssen wir hier ein anderes List-View-Flag benutzen, weil wir diesmal ja die großen (= normalen) Symbole zuweisen wollen:

```
if(hImgBig <> 0) then
  ListView_SetImageList(hLV,hImgBig,LVSIL_NORMAL);
```

So weit die Grundlagen.

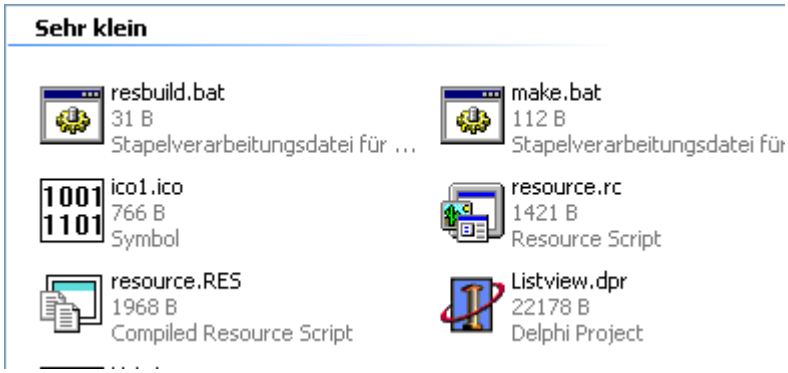
Beim Ermitteln der Symbole erinnern wir uns daran, dass wir uns mit der Funktion "SHGetFileInfo" ja bereits die Typennamen aus dem System geholt haben. Da wir die Funktion auch benötigen, um den Index des jeweiligen Symbols zu ermitteln, liegt es doch eigentlich nahe, beide Aufrufe zusammenzufassen und sowohl Dateityp als auch Icon gleich zu ermitteln. Wir fassen also beides zusammen:

```
ZeroMemory(@fi,sizeof(TSHFileInfo));
SHGetFileInfo(finddata.cFilename,0,fi,sizeof(TSHFileInfo),
  SHGFI_ICON or SHGFI_SYSICONINDEX or SHGFI_TYPENAME);
```

Auf diese Weise erhalten wir den Index des Symbols in der iIcon-Membervariablen und können diesen an unseren List-View-Eintrag weitergeben:

```
lvi.iImage    := fi.iIcon;
```

Mehr muss nicht getan werden, und das Ergebnis ist verblüffend. :o)



Achtung!

Wie bereits angesprochen, dürfen Sie die System-Imagelisten **auf keinen Fall** freigeben! Wenn Sie sich für diesen Weg hier entscheiden, dann ignorieren Sie, dass Ihnen in diesem Kapitel gesagt wurde, wie man eine Imageliste freigibt. Die Folgen sind zwar nur temporärer Natur (ein Neustart, und alles ist wieder beim alten), aber dennoch unangenehm. Bitte denken Sie daran!

3.7.6.3. Umschalten der Ansicht

Das Umschalten der Ansicht gestaltet sich relativ einfach. Man "entfernt" einfach den aktuellen Fensterstil für die Ansicht und setzt einen neuen. Dazu dienen uns zwei Funktionen, "GetWindowLong" und "SetWindowLong". In unserem Fall interessiert uns dabei das "normale" Stilattribut, GWL_STYLE. Mit einer speziellen Prozedur, die als Beispiel im PSDK zu finden ist, können wir uns die Arbeit dabei erleichtern. Wir lesen zuerst den aktuellen Stil aus und ändern die Ansicht nur, wenn der gewünschte Stil noch nicht eingestellt ist:

```
procedure SetView(hWndListView: HWND; dwView: dword);
var
  dwStyle : dword;
begin
  dwStyle := GetWindowLong(hWndListView,GWL_STYLE);

  if(dwStyle and LVS_TYPEMASK <> dwView) then
    SetWindowLong(hWndListView,GWL_STYLE,
      dwStyle and not LVS_TYPEMASK or dwView);
end;
```

Anmerkung
and not stellt das Gegenstück zu **or** da und "löscht/entfernt" ein Attribut.

GetWindowLong-Definition

```
LONG GetWindowLong(
  HWND hWnd,  // Fenster-Handle
  int nIndex  // Attribut, das ausgelesen werden soll
);
```

SetWindowLong-Definition>

```
LONG SetWindowLong(
  HWND hWnd,       // Fenster-Handle
  int nIndex,      // Attribut, das gesetzt werden soll
  LONG dwNewLong   // neuer Wert
);
```

Im Fall unseres Beispielprogramms prüfen wir nun nur noch, welcher Menüeintrag gewählt wurde und rufen diese Prozedur mit dem gewünschten Stil für die Ansicht auf:

```
case LOWORD(wparam) of
  IDM_ICONS:
    SetView(hLV,LVS_ICON);
  IDM_SMICONS:
    SetView(hLV,LVS_SMALLICON);
  IDM_LIST:
    SetView(hLV,LVS_LIST);
  IDM_REPORT:
    SetView(hLV,LVS_REPORT);
end;
```

Hinweis für Windows XP

Ab Windows XP verfügt die List-View über eine eingebaute Funktion, "`ListView_SetView`", mit der sich die Ansicht umschalten lässt. Dazu ist allerdings auch wieder die schon erwähnte Manifest-Datei erforderlich. Nun könnte man meinen, dass der praktische Nutzen gegen Null tendiert, denn mit dieser neuen Funktion können die älteren Windows-Versionen ja nichts anfangen. Aber leider lässt sich nur auf diese Weise die neue "Tile"-Ansicht (Kacheln) unter Windows XP aktivieren. Deshalb müssen wir sie (zumindest unter XP) benutzen.

Ich habe es mir einfach gemacht und o.g. Funktion "`SetView`" erweitert. Wird das Programm unter Windows XP gestartet, dann werden die "alten" Werte benutzt, um die neue Funktion entsprechend aufzurufen. Die Ausnahme bildet nur der schon erwähnte "Kachelmodus", für den es natürlich keine "alte" Entsprechung gibt. Daher habe ich den Wert "666" benutzt.

```
LVS_ICON:
  ListView_SetView(hLV,LV_VIEW_ICON);
LVS_SMALLICON:
  ListView_SetView(hLV,LV_VIEW_SMALLICON);
LVS_LIST:
  ListView_SetView(hLV,LV_VIEW_LIST);
LVS_REPORT:
  ListView_SetView(hLV,LV_VIEW_DETAILS);
666:
  ListView_SetView(hLV,LV_VIEW_TILE);
```

Hinter der Funktion steckt eine neue Nachricht, "`LVM_SETVIEW`", mit ein paar neuen Konstanten. Zu dieser Nachricht gibt es auch ein Gegenstück, "`LVM_GETVIEW`" (bzw. "`ListView_GetView`"), mit dem Sie herausfinden können, welcher Modus gerade aktiv ist.

```
lvMode := ListView_GetView(hLV);
```

3.7.7. Sortieren beim Klick auf den Spaltenkopf

So, jetzt kommt der harte Teil: das Sortieren der Einträge, wenn man auf den Spaltenkopf klickt. Was wollen wir erreichen? Wenn wir eine Spalte auswählen, dann sollen die Einträge in dieser Spalte aufsteigend (A - Z) oder absteigend (Z - A) sortiert werden.
Daher benötigen wir zuerst einmal eine Variable, mit der wir die Sortierrichtung beeinflussen können. Da wir nur zwei Richtungen haben (aufsteigend, absteigend), genügt eine `byte`- oder `bool`-Variable. Das bleibt eigentlich Ihnen überlassen. Ich habe mich für unser Beispiel für eine `byte`-Variable entschieden, die mit dem Wert Null initialisiert wird:

```
var
  SortOrder : byte = 0;
```

Was passiert nun, wenn wir auf einen Spaltenkopf klicken? Die List-View sendet die Benachrichtigung "`LVN_COLUMNCLICK`" (in Form einer "`WM_NOTIFY`"-Nachricht) an unser Hauptfenster. Mitgeliefert werden das Handle der List-View und (in einem `TNMListView`-Record) die angeklickte Spalte. Dieser Wert ist über den `lParam` zugänglich, der auf das besagte Record zeigt.
Ausgehend davon, sieht die Kontrolle in unserem Programm erst einmal so aus:

```
WM_NOTIFY:
  with PNMHdr(lp)^ do
    if(hwndFrom = hLV) then
      case code of
        LVN_COLUMNCLICK:
          begin
            { ... }
          end;
      end;
```

WM_NOTIFY-Definition

```
WM_NOTIFY
    idCtrl = (int) wParam;          // Control, das die Nachricht ausgelöst hat
    pnmh = (LPNMHDR) lParam;        // Zeiger auf eine NMDHR-Strukur
```

NMHDR-Definition

```
typedef struct tagNMHDR {
    HWND hwndFrom;                  // Handle des "Senders"
    UINT idFrom;                    // ID des "Senders"
    UINT code;                      // Benachrichtigungscode
} NMHDR;
```

3.7.7.1. Das Sortieren

Zum Sortieren der Einträge gibt es die Nachricht "`LVM_SORTITEMS`" bzw. das Makro "`ListView_SortItems`". Beiden ist gemeinsam, dass Sie einen anwendungsbezogenen Parameter und die Adresse unserer eigenen Sortierfunktion erwarten. Fangen wir mit letzterer an. Der Name der Funktion ist Ihnen überlassen, lediglich an die Deklaration sollten Sie sich halten:

```
type
  PFNLVCOMPARE = function(lParam1, lParam2, lParamSort: Integer): Integer
stdcall;
```

lParam1, **lParam2**
Die zu vergleichenden Werte

lParamSort
Diesen Parameter geben wir beim Aufruf der o.g. Nachricht oder Funktion an. In vielen Fällen wird hier ein Wert gewählt, der Aussagen über die Sortierrichtung o.ä. trifft. Ich persönlich bin jedoch der Meinung, die Angabe des geklickten Spaltenkopfes ist die bessere Lösung.

Damit habe ich es schon verraten: wenn wir die Sortierung auslösen, dann geben wir den Index der angeklickten Spalte als anwendungsbezogenen Parameter an. Ich sage auch gleich den Grund, aber erst schauen wir uns den Aufruf an. Einmal mit Hilfe der Nachricht:

```
SendMessage(hwndFrom,LVM_SORTITEMS,WPARAM(PNMListView(lp)^.iSubItem),LPARAM(@Com
pareFunc));
```

und einmal mit Hilfe des Makros:

```
ListView_SortItems(hwndFrom,@CompareFunc,PNMListView(lp)^.iSubItem);
```

Nun passiert folgendes: die Index-Werte des ersten und zweiten Eintrags der List-View werden mit dem Index-Wert der angeklickten Spalte an unsere Sortierfunktion übergeben. Mit Hilfe des Spaltenindex entscheiden wir, wie wir sortieren wollen.
Dazu erinnern wir uns an das Beispielprogramm. Es besitzt drei Spalten. Die erste zeigt den Dateinamen, die mittlere zeigt die Dateigröße, und die letzte zeigt den Dateityp an. Das heißt, wir haben zwei String-Werte und einen Integer-Wert. Dementsprechend unterscheidet sich die Sortierung, da wir einen `integer` nicht mit Hilfe eines String-Vergleichs sortieren können.

Doch zuerst holen wir uns erst einmal den Text der beiden Items, die miteinander verglichen werden sollen. Im Beispiel habe ich zwei `string`-Variablen benutzt, die zuerst natürlich initialisiert werden. Wenn das geschehen ist, benutzen wir den Wert von "`lp1`" zum Auslesen des Textes. Der Spaltenindex darf dabei natürlich nicht vergessen werden:

```
ListView_GetItemText(hLV,lp1,SubItem,@buf1[1],MAX_PATH);
```

Angenommen, wir haben die erste Spalte angeklickt, dann wäre "`SubItem`" Null, und da "`lp1`" in dem Fall den Index des allerersten Eintrags enthält, hätten wir mit obiger Anweisung den Dateinamen des ersten Eintrags in der List-View ermittelt.
Das gleiche machen wir dann mit dem zweiten Eintrag, dessen Index in "`lp2`" übergeben wird:

```
ListView_GetItemText(hLV,lp2,SubItem,@buf2[1],MAX_PATH);
```

Wenn wir jetzt davon ausgehen, dass es sich um zwei Strings handelt (also um den Inhalt der Spalten #1 und #3), dann können wir mit der Funktion "`lstrcmpi`" einen recht einfachen Vergleich durchführen. Diese Funktion liefert Null zurück, wenn beide Strings identisch sind. Ist der erste String kleiner als der zweite, dann ist das Ergebnis negativ. Ist der erste String größer als der zweite, liefert die Funktion ein positives Ergebnis zurück.
Kleiner und größer bezieht sich dabei nicht ausschließlich auf die Länge des jeweiligen Strings. Damit sind auch die Zeichen im direkten Vergleich gemeint. Folgende Strings sind identisch und hätten das Ergebnis Null zur Folge:

```
Abcd    = abcd
```

Da wir mit "`lstrcmpi`" unabhängig von der Schreibweise sind, können Sie meine Aussage so akzeptieren. :o)
Folgender Vergleich hätte ein positives Ergebnis zur Folge, denn obwohl der erste String kürzer ist, hat er das "höhere" Zeichen an einer früheren Stelle als der zweite String:

```
abcz     > abcdefg
```

Würden Sie die beiden Strings vertauschen, bekämen Sie ein negatives Ergebnis, da es diesmal der zweite String wäre, der das höherwertige Zeichen zuerst besitzt:

```
abcdefg < abcz
```

Nach dem selben Prinzip vergleichen wir nun unsere beiden Strings. Allerdings möchte ich Sie an unsere Bedingung vom Anfang erinnern: wir wollen auf- oder absteigend sortieren. Das bedeutet, wir müssen unsere oben deklarierte Variable zu Rate ziehen. Wie gehen wir vor? Ganz einfach: wollen wir aufsteigend sortieren (A - Z), dann vergleichen wir das erste Item mit dem zweiten. Wollen wir absteigend sortieren (Z - A), liefern wir die beiden Strings vertauscht an die Funktion und vergleichen sozusagen das zweite mit dem ersten Item:

```
if(SortOrder = 1) then Result := lstrcmpi(@buf2[1],@buf1[1])
  else Result := lstrcmpi(@buf1[1],@buf2[1]);
```

Das eigentliche Sortieren übernimmt nun wieder die List-View für uns. Von unserer Seite waren nur die Werte (negativ < Null < positiv) wichtig, da sie entscheiden, wie die List-View arbeitet. Geben wir einen negativen Wert zurück, dann bedeutet das: das erste Item soll vor dem zweiten eingefügt werden. Geben wir einen positiven Wert zurück, dann bedeutet das: das erste Item soll hinter dem zweiten eingefügt werden. Und der Wert Null bedeutet, dass beide Items identisch sind und nicht sortiert werden.

Nachdem die List-View dies durchgeführt hat, wird die Funktion erneut aufgerufen. Diesmal werden ihr aber die Indexwerte des zweiten und dritten Parameters übergeben ... Auf diese Weise werden alle Einträge der List-View berücksichtigt, und die Sortierfunktion läuft bis es nichts mehr zum Sortieren gibt.

LVM_SORTITEMS-Definition zeigen

```
LVM_SORTITEMS
    wParam = (WPARAM) (LPARAM) lParamSort;       // ein selbst definierter Wert
    lParam = (LPARAM) (PFNLVCOMPARE) pfnCompare; // Adresse der Sortierfunktion
```

3.7.7.2. Halt, da war doch noch was ...

Richtig! Die Integerwerte der mittleren Spalte (die Dateigrößen). Hier wird nun auch klar, warum wir mit "`lstrcmpi`" nicht weit kommen. Ein Vergleich zwischen den Werten 13 und 2 z.B. hätte das Ergebnis, dass die Zwei (im Vergleich mit der Eins als erstem Zeichen der Dreizehn) als größer eingestuft wird.
Aus diesem Grund prüfen wir in unserer Sortierfunktion, welche Spalte angeklickt wurde. Wenn der übermittelte Index den Wert Eins hat, dann entspricht dies der mittleren Spalte mit den Dateigrößen, und wir verzweigen in eine separate Prüfung.

In dieser wandeln wir die beiden Strings in Integer-Werte um. Der Einfachheit halber habe ich dafür die Funktion "`StrToIntDef`" nachgebildet, der gleich noch ein Defaultwert übergeben werden kann. Zu beachten ist allerdings, dass eine Größenangabe in der List-View z.B. so aussehen kann: `26 B`. Wir müssen also alles nach einem evtl. vorhandenen Leerzeichen entfernen, damit wir bei der Konvertierung des Strings in einen numerischen Wert tatsächlich nur die Ziffern berücksichtigen.

Wenn das erledigt ist, denken wir wieder an unsere Sortierrichtung und weisen die Integer-Werte dementsprechend unterschiedlich zu:

```
if(SortOrder = 1) then begin
  b := StrToIntDef(buf1,0);
  a := StrToIntDef(buf2,0);
end else begin
  a := StrToIntDef(buf1,0);
  b := StrToIntDef(buf2,0);
end;
```

Der Vergleich der beiden Integer sollte nun das geringste Problem sein. Ausgehend von der Maßgabe (negativ < Null < positiv) vergleichen wir wie folgt:

```
if(a > b) then Result := 1
  else if(a < b) then Result := -1
    else Result := 0;
```

3.7.7.3. Das reicht aber immer noch nicht!

Wenn Sie die Sortierung so ausprobieren, werden Sie feststellen, das nichts passiert. Warum? - Die Antwort findet sich in der Erklärung des `TLVItem`-Records im PSDK, wo es (frei übersetzt) heißt:

PSDK:

lParam
Item-bezogener Wert. Wenn Sie die Nachricht `LVM_SORTITEMS` verwenden, sendet die List-View diesen Wert an die Sortierfunktion der Anwendung.

Da unsere Items aber beim Füllen keinen Wert für die `lParam`-Membervariable des `TLVItem`-Records zugewiesen bekamen, passiert auch nichts. Wir müssen dies also noch tun, wobei wir ganz einfach nur den jeweiligen Item-Index als `lParam`-Wert festlegen:

```
lvi.mask     := LVIF_PARAM;
lvi.iSubItem := 0;

for i        := 0 to ListView_GetItemCount(hLV) - 1 do begin
  lvi.iItem  := i;
  lvi.lParam := i;
  SendMessage(hLV,LVM_SETITEM,0,LPARAM(@lvi));
end;
```

Im Beispielprogramm finden Sie diese Anweisungen in der Prozedur "`UpdateLParam`", die jeweils vor der Sortierfunktion aufgerufen wird.

3.7.7.4. Die Sortierrichtung umkehren

Bleibt nur eins zu tun: wenn wir aufsteigend sortiert haben, müssen wir dafür sorgen, dass beim nächsten Klick in umgekehrter Reihenfolge sortiert wird ... und wieder umgekehrt, nach einem nochmaligen Klick, usw. Dazu gehen wir zurück zu unserer "`LVN_COLUMNCLICK`"-Auswertung und negieren den aktuellen Wert der Variable "`SortOrder`" einfach:

```
SortOrder := 1 - SortOrder;
```

Jetzt weist unser Programm aber einen Schönheitsfehler auf: egal welchen Spaltenkopf wir anklicken, es wird immer der aktuelle Wert von "`SortOrder`" zur Sortierrichtung herangezogen. Schauen Sie sich das aber mal im Windows Explorer an: wenn Sie dort auf einen Spaltenkopf klicken, wird immer erst aufsteigend (A - Z) sortiert. Erst der nochmalige Klick auf den gleichen Spaltenkopf kehrt die Richtung um.
Es ist daher zweckmäßig, den jeweils aktuellen Spaltenindex zu sichern und im Fall einer Änderung "`SortOrder`" wieder auf Null zu setzen (für die aufsteigende Sortierung), was verkürzt so aussieht:

```
if(LastCol <> PNMListView(lp)^.iSubItem) then
  SortOrder := 0;

{ ... } // sortieren

LastCol   := PNMListView(lp)^.iSubItem;
```

Wichtig!
Während des Sortierprozesses ist der Inhalt der List-View (laut PSDK) "instabil". Nach Möglichkeit sollte also die Arbeit mit der List-View während der Sortierung vermieden werden.

3.7.8. Den "in place"-Editor verwenden

Sie kennen den so genannten "in place"-Editor sicher schon aus anderen Programmen. Der Windows Explorer ist ein gutes Beispiel. Sie klicken einmal auf einen Dateinamen bzw. benutzen die Taste F2, und es erscheint eine Art Eingabefeld, das sich direkt im Control befindet:

Für diese Fähigkeit ist zuerst ein Stilattribut erforderlich, das beim Erzeugen der List-View angegeben werden muss: `LVS_EDITLABELS`:

```
hLV := CreateWindowEx(WS_EX_CLIENTEDGE,WC_LISTVIEW,nil,
  WS_VISIBLE or WS_CHILD or LVS_REPORT or { neu -> } LVS_EDITLABELS or
  LVS_SHOWSELALWAYS,0,0,100,100,wnd,IDC_LV,hInstance,nil);
```

Streng genommen könnte man jetzt sagen: das war´s.
Wirklich!

Die List-View stellt uns den "in place"-Editor bereit, nur nutzt uns das nicht viel - da wir mit ihm nichts weiter anfangen. Es fehlt also noch ein Schritt, und der heißt wie üblich: wir müssen auf eine Nachricht reagieren. Eigentlich sind es sogar zwei: "`LVN_BEGINLABELEDIT`" und "`LVN_ENDLABELEDIT`".

Die Namen sagen es eigentlich schon: "`LVN_BEGINLABELEDIT`" benachrichtigt das Parent-Fenster der List-View, dass mit dem Editieren einer Bezeichnung (= eines Labels) begonnen wird. Demzufolge informiert "`LVN_ENDLABELEDIT`" darüber, dass die Bearbeitung abgeschlossen ist oder abgebrochen wurde. Beide Nachrichten werden via "`WM_NOTIFY`" gesendet.

```
WM_NOTIFY:
  case PNMHdr(lParam)^.code of
    LVN_BEGINLABELEDIT:
      { ... }
    LVN_ENDLABELEDIT:
      { ... }
  end;
```

Der `lParam` ist in beiden Fällen ein Zeiger auf das Record `TLVDispInfo`, wobei uns die `item`-Membervariable interessiert, da Sie den alten bzw. neuen Text unseres Labels enthält. Natürlich enthält sie mehr Infos, aber hauptsächlich geht es uns ja um den Text.

NMLVDISPINFO-Definition

```
typedef struct tagNMLVDISPINFO {
    NMHDR hdr;
    LVITEM item;   // TLVItem-Record
}
```

Folgendes ist zu sagen:

Es ist nicht erforderlich, beide Nachrichten für eine "Vorher-Nachher-Show" der Labelbezeichnung abzufangen. Den neuen Text müssen Sie selbst setzen. Folgendes Codebeispiel würde Ihnen (am Beispiel von "`LVN_ENDLABELEDIT`") die alte und die neue Bezeichnung anzeigen, aber eben nichts in der List-View ändern:

```
LVN_ENDLABELEDIT:
  if(PLVDispInfo(lp)^.item.pszText <> '') then begin
    i := ListView_GetNextItem(hwndFrom,-1,LVNI_FOCUSED);
    if(i > -1) then begin
      ZeroMemory(@buf,sizeof(buf));
      ListView_GetItemText(hwndFrom,i,0,buf,sizeof(buf));

      MessageBox(0,pchar(Format('"%s" vs. "%s"',
        [buf,PLVDispInfo(lp)^.item.pszText])),nil,0);
    end;
  end;
```

Wollen Sie die Bezeichnung nun tatsächlich ändern, geben Sie als Rückgabewert **true** an (oder im Fall einer "WndProc"-Nachrichtenfunktion jeden anderen Wert als Null). Ansonsten muss das Ergebnis **false** (Null) sein

```
      Result := 1;
```

3.7.8.1. Wie war das mit F2?

Angesprochen habe ich es ja bereits: es besteht neben dem Anklicken natürlich auch die Möglichkeit ein Label über ein Menü, einen Button, einen Shortcut, usw. zu ändern. Die List-View kennt dazu die Nachricht "LVM_EDITLABEL", oder Sie verwenden das Makro "ListView_EditLabel", das diese Nachricht kapselt. Das Ergebnis ist das selbe, Sie müssen sich nur für einen Weg entscheiden.

Das Beispielprogramm verwendet sowohl Menü als auch den Shortcut F2 zum Ändern. Sehen wir uns den Menüteil an, denn er enthält den eigentlich produktiven Code. Zuerst sollten wir feststellen, ob überhaupt ein Eintrag ausgewählt ist:

```
i := ListView_GetNextItem(hLV,-1,LVNI_FOCUSED);
if(i > -1) then begin
  { ... }
end;
```

Wenn das der Fall ist ("x" ist in dem Fall größer als -1), dann geben wir der List-View den Fokus:

```
SetFocus(hLV);
```

und rufen entweder die o.g. Nachricht oder das Makro auf. Ich habe mich für das Makro entschieden, aber (wie gesagt!) das bleibt Ihnen überlassen:

```
ListView_EditLabel(hLV,i);
```

Sie geben das Fenster-Handle der List-View und den Wert des selektierten Eintrags an. Bei der Nachricht würde es so aussehen, dass Sie den Wert ("x") als wParam angeben müssen:

```
SendMessage(hLV,LVM_EDITLABEL,i,0);
```

Beim Shortcut schreiben wir nun nicht alles noch einmal; wir generieren stattdessen eine Klick-Nachricht und rufen damit den Menücode auf:

```
SendMessage(wnd,WM_COMMAND,MAKEWPARAM(IDM_EDITLABEL,BN_CLICKED),0);
```

Alternative zum Shortcut

Wenn Sie keinen Shortcut benutzen wollen, steht Ihnen auch noch die List-View selbst zur Verfügung. Wenn Sie die Nachricht "LVN_KEYDOWN" bearbeiten, können Sie ebenfalls auf die Taste F2 reagieren. Das Ergebnis ist das gleiche, nur Sie sparen den Shortcut ein.

"LVN_KEYDOWN" wird als Teil von "WM_NOTIFY" gesendet und muss dementsprechend bearbeitet werden:

```
WM_NOTIFY:
  case PNMHdr(lp)^.code of
    LVN_KEYDOWN:
      { ... }
  end;
```

Den Tastencode kommen wir dann mit einem Zeiger auf eine `TLVKeyDown`-Variable, wobei `wvKey` die benötigte Membervariable ist:

```
if(PLVKeyDown(lp)^.wvKey = VK_F2) then
  { ... }
```

Da es sich aber (wie gesagt!) um einen Zeiger handelt, verwenden wir in dem Fall also `PLVKeyDown`.

TLVKEYDOWN-Definition zeigen

```
typedef struct tagLVKEYDOWN {
    NMHDR hdr;    // weitere Infos zur Nachricht
    WORD wVKey;   // virtueller Tastencode
    UINT flags;   // ist immer Null
}
```

3.7.9. Einträge automatisch markieren

Wir wollen uns nun kurz ansehen, wie man Einträge programmgesteuert markieren kann und wie man ihre Markierung umkehrt.
Wozu soll das gut sein?
Ein typisches Beispiel wäre ein Dateimanager. Sie haben die Möglichkeit eine Datei auszuwählen, ein paar Dateien ... oder eben auch alle. Und wenn Sie z.B. alle Dateien bis auf eine bestimmte löschen wollen, dann ist es eindeutig schneller, wenn Sie diese eine Datei auswählen und diese Auswahl dann quasi umdrehen.

Um Ärger zu vermeiden (derart, dass Sie denken: "Hey, das funktioniert irgendwie nicht!"), spendieren wir unserem List-View-Control ein neues Stilattribut:

```
hLV := CreateWindowEx(WS_EX_CLIENTEDGE,WC_LISTVIEW,nil,
  WS_VISIBLE or WS_CHILD or LVS_SHOWSELALWAYS, { ... } );
```

Das Attribut `LVS_SHOWSELALWAYS` sorgt nur dafür, dass die aktuelle Auswahl immer sichtbar bleibt, egal ob die List-View den Fokus hat oder nicht.

Nun aber zum eigentlichen Thema -

Unser Beispiel hat zwei neue Menüpunkte, "Alles markieren" und "Markierung umkehren". Beide rufen die selbe Prozedur auf, denn das Prinzip ist das selbe; nur ein `bool`-Parameter entscheidet, wie sich die Prozedur verhält. In dieser Prozedur wird zunächst der aktuelle Status jedes Eintrags abgefragt. Ich habe mich für die Benutzung des Makros "`ListView_GetItemState`" entschieden, was aber eigentlich nur eine Kapselung der Nachricht "`LVM_GETITEMSTATE`" ist. In beiden Fällen wird der Index des jeweiligen Eintrags benötigt. Dazu kommt ein Flag (oder eine Kombination mehrerer Flags), damit wir auch erfahren was uns interessiert! Der Aufruf sieht also so aus:

```
uRes := ListView_GetItemState(hLV,i,LVIS_SELECTED);
```

"`i`" ist dabei der Index des jeweiligen Eintrags, und `LVIS_SELECTED` sagt aus, dass wir uns für die Markierung des Eintrags interessieren. Weiterhin möglich wären:

```
LVIS_CUT
LVIS_DROPHILITED
LVIS_FOCUSED
LVIS_OVERLAYMASK
LVIS_STATEIMAGEMASK
```

Diese Flags müssten, wenn Sie mehrere nutzen wollen, **or**-verknüpft werden, etwa

```
uRes := ListView_GetItemState(hLV,i,LVIS_SELECTED or LVIS_FOCUSED);
```

Das bedeutet aber auch, dass Sie für die Prüfung nicht einfach auf Gleichheit testen können. Wir müssen also herausfinden, ob das Flag, das uns interessiert, im Rückgabeergebnis des Makros enthalten ist. Aber eigentlich interessiert uns ja, ob das Flag nicht im Rückgabeergebnis enthalten ist, also:

```
if(uRes and LVIS_SELECTED = 0) then ;
```

Am Rande
Wie würde eigentlich der Aufruf der o.g. Nachricht aussehen, wenn wir nicht das Makro verwenden wollen? Nun, einfach so:

```
uRes := SendMessage(hLV,LVM_GETITEMSTATE,i,LVIS_SELECTED);
```

Um den Status eines Eintrags zu ändern, verwenden wir das Gegenstück zu obigem Makro: "`ListView_SetItemState`". Das Makro erwartet zunächst (abgesehen vom Handle der List-View) wieder den Index des jeweiligen Eintrags. Der dritte Parameter entscheidet, ob in unserem Fall die Markierung gesetzt oder aufgehoben werden soll. Der vierte Parameter ist wieder das Flag, das uns interessiert (die Markierung eben!).

```
ListView_SetItemState(hLV,i,LVIS_SELECTED,LVIS_SELECTED);
```

Mit dieser Anweisung würde ein Eintrag markiert werden. Um seine Markierung aufzuheben, verwenden Sie als dritten Parameter eine Null anstelle von `LVIS_SELECTED`:

```
ListView_SetItemState(hLV,i,0,LVIS_SELECTED);
```

In unserem Fall ist also entscheidend, ob ein Item markiert ist oder nicht. Ist es nicht markiert, wird es markiert. Ist es markiert, wird diese Auswahl aufgehoben. Nach diesem Prinzip arbeitet der Befehl "Markierung umkehren". Bei der Funktion "Alles markieren" wird der ermittelte Status ignoriert und generell die Markierung gesetzt.

Folgendes ist anzumerken:
Das selbe passiert natürlich auch im Beispielprogramm. Allerdings sieht es dort evtl. etwas merkwürdig aus, da ich eine **if**-Abfrage eingespart habe. Ich benutze stattdessen ein `bool`-Array:

```
const
  fStateState : array[boolean]of cardinal = (0,LVIS_SELECTED);
```

Dieses Array hat nur zwei Zustandsformen: **true** (= `LVIS_SELECTED`) oder **false** (= 0). Anstelle eine umständliche Abfrage zu schreiben wie:

```
if(uRes and LVIS_SELECTED = 0) or (not fTurnSelection) then
  ListView_SetItemState(hLV,i,LVIS_SELECTED,LVIS_SELECTED)
else
  ListView_SetItemState(hLV,i,0,LVIS_SELECTED);
```

übergebe ich dem Array lediglich den ermittelten Status des Eintrags und die `bool`-Variable aus dem Prozedurkopf als **or**-Verknüpfung:

```
ListView_SetItemState(hLV,i,
  fStateState[(uRes and LVIS_SELECTED = 0) or (not fTurnSelection)],
  LVIS_SELECTED);
```

Das Ergebnis ist das selbe, nur die **if**-Verzweigung entfällt.

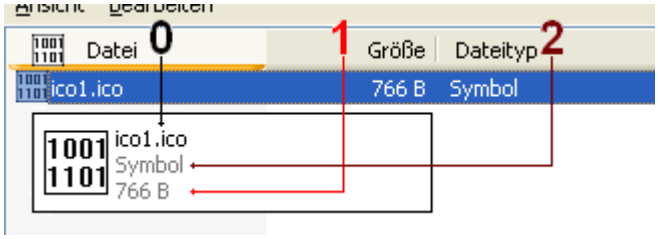
Probieren Sie es einfach aus!
Starten Sie das Beispielprogramm und markieren Sie z.B. das erste Item, das dritte, das fünfte, usw. Dann benutzen Sie die Funktion "Markierung umkehren". Oder markieren Sie gar nichts, und wählen Sie dann "Alles markieren".

3.7.10. Die "Tile"-Ansicht der List-View unter Windows XP

Wenn wir über die verschiedenen Anzeigemodi der List-View sprechen, dann dürfen wir die neue "Tile"-Ansicht von Windows XP nicht außer Acht lassen. Dabei ist, ähnlich wie in der Icon-Ansicht, das Symbol des jeweiligen Eintrags zu sehen. Neben diesem Symbol, und das ist der Unterschied, sehen Sie den Namen des Eintrags und eine oder mehrere zusätzliche Zeilen mit weiterem Text.
Für diese neue Ansicht ist auch wieder eine Manifestdatei erforderlich, die dem Programm entweder beiliegen kann, bzw. die in den Ressourcen integriert ist.

Wenn Ihre Delphi-Version die benötigten Records und Funktionen, auf die wir noch eingehen werden, nicht kennt, dann können Sie die beiliegende "`CommCtrl_Fragment.pas`" benutzen. Diese enthält alle notwendigen Informationen.

Wo beginnen wir nun?
Am besten überlegen wir uns vorher, welche Informationen wir anzeigen wollen. Dass die Textzeilen im "Tile"-Modus mit einigen, ausgewählten Spalten (aus dem Report-Modus) identisch sind, möchte ich anhand eines Screenshots verdeutlichen:



Von Interesse sind hierbei eigentlich nur die Spalten #2 und #3 (im Bild **1** und **2**; wg. der null-basierenden Zählweise). Der Inhalt der ersten Spalte, identisch mit dem Hauptnamen des Items, wird nämlich in jedem Fall und in jedem Darstellungsmodus angezeigt, so dass wir ihn auch ignorieren können.

3.7.10.1. Schritt #1 - Grundlegende Einstellungen für die List-View

Wenn wir den "Tile"-Modus verwenden wollen, müssen wir zunächst einmal ein paar generelle Einstellungen treffen. Andernfalls werden standardmäßig nur zwei Zeilen (also der Hauptname und eine zusätzliche Zeile) angezeigt.
Das hierfür benötigte Record ist vom Typ `TLVTileViewInfo` und muss mit seiner Größe initialisiert werden

```
tileinfo.cbSize  := sizeof(TLVTileViewInfo);
```

Da wir die Anzahl der Zeilen ändern wollen, müssen wir die `dwMask`-Membervariablen entsprechend setzen:

```
tileinfo.dwMask  := LVTVIM_COLUMNS;
```

Als Flag erscheint mir der Wert `LVTVIF_AUTOSIZE` ideal. Damit passen sich die einzelnen Tiles den Abmessungen des Fensters an, besitzen aber eine Mindestbreite

```
tileinfo.dwFlags := LVTVIF_AUTOSIZE;
```

In der `cLines`-Membervariablen geben wir nun die Anzahl der zusätzlichen Zeilen für den "Tile"-Modus an. Der Hauptname des Items ist immer sichtbar, zählt also nicht mit:

```
tileinfo.cLines  := 2;
```

Das so gefüllte Record reichen wir nun mit Hilfe der Nachricht "LVM_SETTILEVIEWINFO" oder der Funktion "ListView_SetTileViewInfo" an die List-View weiter:

```
ListView_SetTileViewInfo(hLV,tileinfo);
```

Feste Größen
Sie können die Größe einer solchen "Kachel" auch selbst definieren. Dazu ergänzen Sie die Maske um das Flag LVTVIM_TILESIZE, lassen die dwFlags-Membervariable weg und geben in tileSize die gewünschte Größe an; beispielsweise:

```
tileinfo.dwMask      := LVTVIM_COLUMNS or LVTVIM_TILESIZE;
tileinfo.cLines      := 2;
tileinfo.sizeTile.cx := 300;
tileinfo.sizeTile.cy := 70;
```

3.7.10.2. Schritt #2 - Die einzelnen Items berücksichtigen

Würden wir die "Tile"-Ansicht jetzt testen, würden wir nur das Symbol und den Itemnamen rechts daneben sehen. Das liegt daran, dass die List-View bisher noch nicht weiß, wie es mit den einzelnen Items zu verfahren hat, wenn der neue Modus gewählt wird.
Uns stehen nun zwei Wege zur Verfügung, dies zu ändern; sprich: der List-View mitzuteilen, welche Informationen uns im "Tile"-Modus interessieren. Beide Wege erfordern eine Erweiterung der Routine, mit der wir die List-View gefüllt haben.

Das erweiterte TLVItem-Record

Für diesen Weg ist zuvor aber noch eine Ergänzung des TLVItem-Records gemäß den Angaben im PSDK erforderlich. Sie müssen einfach nur die drei rot gekennzeichneten Membervariablen in der Deklaration des Typs (in der Unit "CommCtrl.pas") hinzufügen:

```
type
  tagLVITEMA = packed record
    mask: UINT;
    iItem: Integer;
    iSubItem: Integer;
    state: UINT;
    stateMask: UINT;
    pszText: PAnsiChar;
    cchTextMax: Integer;
    iImage: Integer;
    lParam: LPARAM;
    iIndent: Integer;
    iGroupId: integer;
    cColumns: uint;
    puColumns: puint;
  end;
```

(Bei der Unicode-Version, "tagLVITEMW", ist ebenso zu verfahren.) Wenn Sie dies getan haben, dann schauen Sie sich bitte noch einmal an, wie Ihr erstes Item in die List-View eingefügt wird. Damit auch die Einstellungen für "Tile"-Modus berücksichtigt werden, erweitern wir die Maske um das neue Flag LVIF_COLUMNS (deklariert in der "CommCtrl_Fragment.pas"):

```
lvi.mask       := LVIF_TEXT or LVIF_IMAGE or LVIF_COLUMNS;
```

Dann benutzen wir eine der neuen Membervariablen, cColumns und geben die Anzahl der zusätzlichen Zeilen an, die (zusätzlich zum Hauptnamen des Items) im "Tile"-Modus angezeigt werden sollen:

```
lvi.cColumns  := 2;
```

Hier bietet sich übrigens die erste Eingriffsmöglichkeit. Sie können bestimmte Einträge anders darstellen lassen, indem Sie einen anderen Wert eintragen. Nehmen wir als Beispiel an, jede Datei, deren Indexwert eine ungerade Zahl ist, soll nur eine zusätzliche Zeile anzeigen, dann ließe sich das so realisieren:

```
if(odd(Loop)) then lvi.cColumns := 1
  else lvi.cColumns := 2;
```

Ebenfalls sehr wichtig für die Darstellung ist die Membervariable puColumns. Sie ist ein Zeiger auf ein Integer-Aray, das die Indexwerte der Spalten enthalten muss, die wir im "Tile"-Modus sehen wollen!
Wenn Sie sich das Bild am Anfang ansehen, dann sind das die Spalten **1** und **2** (null-basierende Indexzählweise), wobei der Dateityp der letzten Spalte noch vor der Dateigröße angezeigt werden soll. Unser Array wird daher wie folgt deklariert:

```
var
  colArray : array[0..1]of integer = (2,1);
```

und an die Membervariable übergeben:

```
lvi.puColumns := @colArray[0];
```

Das ist auch schon das ganze Geheimnis. Hätten wir noch mehr Spalten, ließe sich das Array entsprechend vergrößern. Da das Item dann wie gewohnt mit der Nachricht "LVM_INSERTITEM" bzw. der Funktion "ListView_InsertItem" in die List-View aufgenommen wird, müssen wir weiter nichts tun. Schalten wir jetzt in den neuen Ansichtsmodus um, dann sehen wir ein Ergebnis wie auf dem Screenshot am Anfang.

Übrigens: das Array selbst bietet die zweite Eingriffsmöglichkeit in die Darstellung. Wenn Sie sich an die Spielerei mit den ungeraden Indexwerten erinnern: etwas Vergleichbares ließe sich auch hier machen; etwa, dass alle Dateien mit ungeradem Index zuerst die Größe und dann den Dateityp anzeigen, usw. usw.
Experimentieren Sie einfach ein wenig!

Die zweite Variante

Der eben gezeigte Weg hat einen Nachteil: Sie benötigen den Quellcode der Unit "CommCtrl.pas" zum Erweitern des TLVItem-Records. Natürlich ließe sich dieser neue Typ auch in einer externen Datei deklarieren (als TLVItem60 finden Sie ihn übrigens in der "CommCtrl_Fragment.pas"), dann könnten Sie allerdings nicht mehr auf die Makro-Funktionen, wie etwa "ListView_InsertItem" usw., zugreifen, da diese nach wie vor die Originalstruktur von TLVItem benutzen.

Aus dem Grund bin ich im Beispielprogramm einen anderen Weg gegangen: Nachdem ein Item eingefügt wurde, benutzen wir ein weiteres neues Record (Typ: TLVTileInfo). Dieses Record wird ebenfalls initialisiert:

```
tile.cbSize    := sizeof(TLVTileInfo);
```

Dann geben wir den Index des eben eingefügten Items an. Dieser Index entspricht im Beispielprogramm der Variablen "Loop" aus der Prozedur "GetFiles":

```
tile.iItem     := Loop;
```

Die zwei nächsten Parameter und ihre Bedeutung sind bereits vom erweiterten `TLVItem`-Record bekannt. Wir geben also auch hier an, wie viele zusätzliche Zeilen wir im "Tile"-Modus sehen wollen

```
tile.cColumns := 2;
```

und wir setzen die Referenz auf unser Array mit den Indexwerten der anzuzeigenden Spalten:

```
tile.puColumns := @colArray[0];
```

Nun müssen wir diese Einstellungen noch an die List-View übergeben, damit sie auch Beachtung finden:

```
ListView_SetTileInfo(hLV,tile);
```

Und das war´s.

3.7.11. Die Gruppierung von Items unter Windows XP

Ebenfalls ausschließlich Windows XP vorbehalten ist die neue Gruppenansicht der List-View. Das bedeutet, Sie können Einträge zu logischen Gruppen zusammenfassen, die auch räumlich voneinander getrennt sind. Im Vergleich zur normalen Sortierung halte ich das für eine recht nützliche Erweiterung:

(links: Whistler Beta 1 mit der damals noch vorhandenen Anzahl der Einträge)

Die Gruppenansicht kann in den Modi Icons, Small-Icons, Details und "Tiles" benutzt werden. Nur in der normalen Listenansicht (`LVS_LIST`) wird sie vom System deaktiviert. Die Manifestdatei ist ebenfalls erforderlich!

Dass Sie sich um die Gruppen selbst kümmern müssen, mag am Anfang wie ein Nachteil erscheinen. Auf der anderen Seite haben Sie dadurch den Vorteil, dass Sie die Bezeichnungen der Gruppen auf Ihre Anwendung zuschneiden können. Das Beispielprogramm etwa imitiert den Explorer von Windows XP:
Im Normalfall sehen Sie zuerst eine alfabetische Sortierung der Dateinamen. Der Screenshot rechts zeigt Ihnen aber auch noch eine andere Möglichkeit: Wenn Sie die Items nach ihren Typen sortieren lassen, dann werden die Gruppen anhand dieser Typennamen gebildet, und die Dateien werden neu zugeordnet. In der alfabetischen Sortierung würden Sie daher etwa die "`make.bat`" unter **M** und die "`resbuild.bat`" unter **R** finden. In der Typensortierung stecken beide in der gleichen Gruppe.
Und zu guter Letzt gibt es da auch noch die Sortierung nach der Dateigröße, die ebenfalls Auswirkungen auf die Gruppennamen hat.

3.7.11.1. Bevor wir Gruppen bilden ...

... sollten wir dafür sorgen, dass schon vorhandene Gruppen entfernt werden. Das hat damit zu tun, dass jede Gruppe eine eigene ID besitzt und von der List-View gespeichert wird. Um nicht unnötig Ressourcen zu vergeuden, entfernen wir deshalb alle Gruppen sobald sich die Auswahl der Sortierung ändert! Die Neubildung der Gruppen ist nicht erforderlich, wenn Sie die selbe Spalte neu sortieren lassen. Erst wenn Sie z.B. von den Dateinamen zu den Dateitypen wechseln, sollten Sie die Gruppen neu erstellen.
Zum Entfernen der Gruppen nutzen wir die Nachricht "`LVM_REMOVEALLGROUPS`" oder das Makro "`ListView_RemoveAllGroups`":

```
ListView_RemoveAllGroups(hLV);
```

3.7.11.2. Unsere erste Gruppe

Als Beispiel für unsere erste Gruppe möchte ich die alfabetische Sortierung demonstrieren. Hierbei werden lediglich die Anfangsbuchstaben der Items der List-View benutzt. Wir benötigen also zuerst eine **for**-Schleife, mit der wir die einzelnen Items ansteuern können. Und weil der entsprechenden Prozedur, "RebuildGroups", der Index der Spalte übermittelt wird, gestaltet sich das Auslesen des Namens recht einfach:

```
ListView_GetItemText(hLV,i,iSubItem,@buf[1],MAX_PATH);
```

Nun kürzen wir den String auf ein Zeichen und wandeln dieses in den entsprechenden Großbuchstaben um:

```
SetLength(buf,1);
buf[1] := UPCASE(buf[1]);
```

Ohne zu weit vorgreifen zu wollen: Wir arbeiten diesmal mit Unicode-Zeichen; das heißt, alle String-Referenzen müssen in widestring- bzw. pwidechar-Typen konvertiert werden. Ich habe zu dem Zweck einen entsprechenden Textpuffer deklariert

```
var
  wbuf : array[0..MAX_PATH]of widechar;
```

der nun den geplanten Gruppennamen aufnehmen soll. Dabei ist die Frage: Liegt das übrig gebliebene Zeichen unseres Strings im Bereich von A bis Z? Wenn ja, benutzen wir es als Gruppennamen. Wenn nein, dann definieren wir die Gruppe "Andere":

```
if(buf[1] in['A'..'Z']) then lstrcpyW(wbuf,pwidechar(widestring(buf)))
  else lstrcpyW(wbuf,'Andere');
```

Nun ist die Frage: Gibt es diese Gruppe vielleicht schon? Wenn ja, dann müssen wir sie nicht noch einmal erzeugen, sondern dann ist wichtig, dass wir ihre ID ermitteln, damit wir das Item entsprechend zuordnen können. Und damit machen wir Bekanntschaft mit dem neuen TLVGroup-Record, das wir erst einmal zum Auslesen des Gruppennamens benutzen wollen. Da wir die Gruppen nur mit ihrer ID eindeutig bestimmen können, benötigen wir auch noch eine Index-Variable, "gi". Diese Variable wird vor dem Auslesen der Gruppen auf Null gesetzt und bildet mit der minimalen Gruppen-ID (MIN_GROUP_ID = 2000) die jeweilige ID.
Unser Record wird dann initialisiert:

```
ZeroMemory(@group,sizeof(TLVGroup));
group.cbSize := sizeof(TLVGroup);
```

Da wir am Namen interessiert sind, setzen wir die Maske wie folgt:

```
group.mask   := LVGF_HEADER;
```

Mit Hilfe der Nachricht "LVM_GETGROUPINFO" bzw. der Funktion "ListView_GetGroupInfo" können wir dann die gewünschten Informationen auslesen. Hierbei ist auch das Funktionsergebnis von Interesse. Da wir uns in einer Endlosschleife (**while**-Typ, s. Beispielprogramm) befinden, sollten wir diese abbrechen können! Die Abbruchbedingung wäre in diesem Fall der Wert -1, der von der Nachricht und der Funktion zurückgeliefert wird, wenn auf die gewünschte Gruppe nicht zugegriffen werden konnte:

```
if(ListView_GetGroupInfo(hLV,MIN_GROUP_ID+gi,group) = -1) then break;
```

War das Auslesen erfolgreich, enthält die Membervariable pszHeader nun den Namen der Gruppe, den wir mit unserem geplanten Namen vergleichen können. Stimmen beide überein, dann verlassen wir die Schleife, weil wir die ID ja nun haben.

```
  if(lstrcmpiW(wbuf,group.pszHeader) = 0) then begin
    fFound := true;
    break;
  end;
```

Nehmen wir nun einmal an, die Gruppe existiert noch nicht. In dem Fall legen wir sie an, wozu wir wieder das `TLVGroup`-Record benutzen:

```
ZeroMemory(@group,sizeof(TLVGroup));
group.cbSize     := sizeof(TLVGroup);
group.mask       := LVGF_HEADER or LVGF_GROUPID;
```

Sie sehen hier als zusätzliches Flag `LVGF_GROUPID`. Damit wird die Membervariable `iGroupId` des Records wichtig und muss von uns mit einer ID gefüllt werden. Die ID ergibt sich, wie schon angesprochen, aus einem minimalen Wert und dem ermittelten oder (falls es die Gruppe noch nicht gibt) dem Originalwert der Variable "`gi`":

```
group.iGroupId   := MIN_GROUP_ID + gi;
```

Danach weisen wir unseren Gruppennamen und dessen Länge zu

```
group.pszHeader := wbuf;
group.cchHeader := lstrlenW(wbuf);
```

und fügen die Gruppe mit Hilfe der Nachricht "`LVM_INSERTGROUP`" oder dem Makro "`ListView_InsertGroup`" ein:

```
ListView_InsertGroup(hLV,-1,group);
```

Der Wert -1 bedeutet hierbei, dass die Gruppe an das Ende der internen Liste gesetzt werden soll.

Auf diese Weise werden nun alle benötigten Gruppen erzeugt. Wie Sie in Ihrem Programm verfahren, das hängt ganz von der Art Ihrer Anwendung ab. Das Beispielprogramm verwendet (wie schon gesagt) auch die Typennamen zur Gruppierung. Lassen Sie die Dateien nach ihrer Größe sortieren, dann verwendet es sechs verschiedene Schwellenwerte, um Gruppen von "Gleich Null" bis "Sehr groß" bilden zu können. Diese Arbeit nimmt Ihnen das System leider nicht ab, dafür können Sie die Gruppierung flexibel an Ihr Programm anpassen.

Gruppentitel ausrichten

Sie können den Titel der Gruppe auch zentriert oder rechtsbündig ausgeben. Dazu ergänzen Sie in der `mask`-Variablen das Flag `LVFG_ALIGN`

```
group.mask       := { ... } or LVGF_ALIGN;
```

und dann stehen Ihnen die Konstanten `LVGA_HEADER_CENTER` und `LVGA_HEADER_RIGHT` zur Angabe in der `uAlign`-Membervariablen zur Verfügung. (`LVGA_HEADER_LEFT` natürlich auch, aber die linksbündige Auswahl ist ohnehin die Voreinstellung.)

```
group.uAlign     := LVGA_HEADER_CENTER;
```

3.7.11.3. Das Item zuordnen

Bleibt nur noch eins zu tun: das jeweilige Item muss der Gruppe zugeordnet werden. Hierfür benötigen wir allerdings das erweiterte `TLVItem`-Record, das ich bei der ">Tile"-Ansicht bereits angesprochen habe.
Lassen Sie mich das erklären: Ich hatte ursprünglich gedacht, dass die Nachricht "`LVM_MOVEITEMTOGROUP`" (respektive "`ListView_MoveItemToGroup`") benutzt wird, um ein Item einer Gruppe zuzuordnen. Aber entweder dient diese Nachricht einem anderen Zweck, oder sie funktioniert schlichtweg nicht. (@Microsoft: ?) Keiner meiner Versuche war von Erfolg gekrönt.

Nun bietet aber das erweiterte `TLVItem`-Record eine Möglichkeit über die Membervariable `iGroupId`.

Falls Ihre Delphi-Version diese Variable nicht kennt, benutzen Sie bitte den Typ `TLVItem60` aus der Unit "`CommCtrl_Fragment.pas`. Leeren Sie Ihre Variable und weisen Sie als Maske nur das neue Flag `LVIF_GROUPID` zu:

```
ZeroMemory(@lvi,sizeof(lvi));
lvi.mask     := LVIF_GROUPID;
```

Dann übergeben Sie den Index des Items und die ID der Gruppe, in die das Item aufgenommen werden soll:

```
lvi.iItem    := i;
lvi.iGroupId := MIN_GROUP_ID + gi;
```

Und mit Hilfe von "`LVM_SETITEM`" bzw. "`ListView_SetItem`" (sofern Ihre Delphi-Version aktuell genug ist und das erweiterte Record benutzt) weisen Sie dem Item die Gruppen-ID zu:

```
SendMessage(hLV,LVM_SETITEM,0,LPARAM(@lvi));
```

3.7.11.4. Die Gruppen-Ansicht ein- oder ausschalten

Die API stellt uns hierfür die die Nachricht "`LVM_ENABLEGROUPVIEW`" zur Verfügung. Sie erwartet als `wParam` den gewünschten Status: **true** (Ansicht aktivieren) oder **false** (Ansicht deaktivieren)

```
SendMessage(hLV,LVM_ENABLEGROUPVIEW,WPARAM(true),0);
```

Alternativ dazu gibt es die Funktion "`ListView_EnableGroupView`":

```
ListView_EnableGroupView(hLV,true);
```

3.7.12. Die Sortierrichtung im Spaltenkopf anzeigen

Als kleine Ergänzung zur Sortierfunktion wollen wir noch die aktuelle Richtung im Spaltenkopf der List-View anzeigen. Dazu ist zu sagen, dass wir ein wenig auf das Header-Control eingehen müssen, zu dem es kein Tutorial gibt. Wir beschränken uns aber auf das Wesentliche.

Außerdem ist zu sagen, dass wir ab Version 6.0 der Common Controls Bitmaps aus dem System nutzen können. Da aber das Programm evtl. auch auf älteren Windows-Versionen laufen soll, nehmen wir also zunächst unser bevorzugtes Zeichenprogramm (Paint reicht vollkommen!) und gestalten eine Bitmap mit zwei Pfeilen gleicher Breite und Höhe für die Richtungen Aufwärts und Abwärts.

Um herauszufinden, ob wir diese Bitmap überhaupt benötigen, holen wir uns das Handle auf den Header der List-View und senden die Nachricht "`CCM_GETVERSION`":

```
hHeader    := ListView_GetHeader(hLV);
iHeaderVer := SendMessage(hHeader,CCM_GETVERSION,0,0);
```

Ist der Rückgabewert größer oder gleich 6, können wir auf unsere selbst gezeichnete Grafik verzichten. Wir wollen aber davon ausgehen, dass der Wert kleiner als 6 ist. In dem Fall laden wir die Grafik als Imageliste aus den Ressourcen des Programms:

```
hSortImg :=
ImageList_LoadBitmap(hInstance,MAKEINTRESOURCE(BMP_SORTBMP),7,1,$00c0c0c0);
```

und übergeben sie an das Header-Control:

```
Header_SetImageList(hHeader,hSortImg);
```

Von besonderem Interesse ist nach meiner Ansicht der oben rot markierte Wert `$00c0c0c0`. Dieser Wert ergibt sich aus der Hintergrundfarbe meines Beispielbildes: Hellgrau. Durch die Angabe dieses Wertes wird der Hintergrund des Bildes transparent dargestellt. Würden Sie stattdessen einen anderen Wert benutzen, könnten Sie den hellgrauen Hintergrund

sehen. Wenn Sie eine andere Hintergrundfarbe in Ihren Grafiken verwenden, dann müssen Sie natürlich deren Wert angeben!

ImageList_LoadBitmap-Definition zeigen

```
HIMAGELIST ImageList_LoadBitmap(
    HINSTANCE hi,     // Instanzen-Handle
    LPCTSTR lpbmp,    // Bitmap-ID
    int cx,           // Image-Breite
    int cGrow,
    COLORREF crMask
);
```

3.7.12.1. Bitmap aus dem Header entfernen

Auch wenn es sich komisch anhört, zuerst wollen wir eine Grafik aus dem Header entfernen. Das hat nämlich mit unserer Sortierung zu tun. Klicken wir auf den Spaltenkopf, wird ermittelt, ob es sich dabei um eine andere Spalte handelt als bei der letzten Sortierung. Ist das der Fall, dann muss natürlich auch die Bitmap entfernt werden.

Das geht relativ einfach mit dem Makro "Header_GetItem", mit dem wir zunächst alle aktuellen Daten der Spalte ermitteln. Uns interessieren dabei fast alle Parameter, weil wir zwar die Angaben zum Image entfernen, die übrigen Einstellungen aber nicht ändern wollen. Darum sieht die Maske des THDItem-Records so aus:

```
hi.Mask       := HDI_BITMAP or HDI_FORMAT or HDI_IMAGE or HDI_ORDER or
  HDI_TEXT or HDI_WIDTH;
```

Damit der Spaltentext auch gesichert werden kann, übergeben wir noch einen Puffer und dessen Größe

```
hi.pszText    := buf;
hi.cchTextMax := sizeof(buf);
```

und rufen dann "Header_GetItem" auf:

```
Header_GetItem(hwndHeader,iIdx,hi);
```

Da uns die Werte an sich aber nicht interessieren, widmen wir uns lediglich der fmt-Membervariablen und entfernen alles, was irgendwie mit Bitmaps zu tun hat:

```
hi.fmt        := hi.fmt and not HDF_SORTUP  // Pfeil-nach-oben  (v6.0)
  and not HDF_SORTDOWN                      // Pfeil-nach-unten (v6.0)
  and not HDF_BITMAP_ON_RIGHT               // Bitmap-Position
  and not HDF_IMAGE;
```

In den beiden ersten Zeilen sehen Sie die Angaben HDF_SORTUP und HDF_SORTDOWN, die eigentlich nur für neuere Windows-Versionen interessant sind. Aber wir sparen uns damit eine Versionsprüfung. :o)
Das so veränderte Item geben wir nun mit der Funktion "Header_SetItem" an das Header-Control zurück

```
Header_SetItem(hwndHeader,iIdx,hi);
```

und haben eine evtl. vorhandene Grafik damit entfernt.

3.7.12.2. Bitmaps im Header anzeigen

Das Anzeigen einer Grafik entspricht prinzipiell dem eben gezeigten Weg. Zunächst ermitteln wir die aktuellen Daten der Header-Spalte mit "Header_GetItem". Wenn wir diese Daten haben, dann ergänzen wir in der Formatvariablen fmt zunächst, dass wir die Grafik rechts vom Spaltentext anzeigen lassen wollen:

```
hi.fmt        := hi.fmt or HDF_BITMAP_ON_RIGHT;
```

Die Grafik selbst wird versionsabhängig angezeigt. Wenn, wie erwähnt, unsere Windows-Version aktuell genug ist und ein Header-Control der Version 6.0 oder aktueller enthält, dann können wir Bitmaps benutzen, die sich bereits im System

befinden. In dem Fall brauchen wir unsere Imageliste nicht, sondern wir verwenden das Flag `HDF_SORTUP` oder `HDF_SORTDOWN` - abhängig von der Sortierrichtung:

```
if(iHeaderVer >= 6) then hi.fmt := hi.fmt or fSortBmp[SortOrder=0]
```

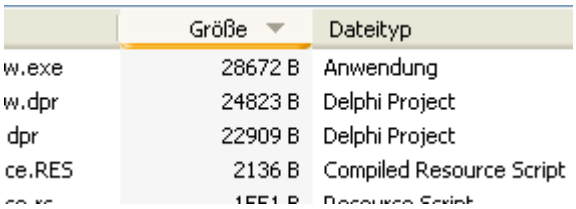
"`fSortBmp`" ist dabei ein `boolean`-Array, das die beiden Flags enthält. Ist die Bedingung erfüllt (`SortOrder = 0`), dann wird `HDF_SORTUP` verwendet, andernfalls `HDF_SORTDOWN`. Bei einer älteren Windows-Version greifen wir stattdessen auf unsere Imageliste zu und setzen zunächst das Flag `HDF_IMAGE`. Damit wird das Header-Control angewiesen, eine Grafik aus der anfangs zugewiesenen Imageliste zu verwenden.

```
  else begin
    hi.fmt    := hi.fmt or HDF_IMAGE;
```

Im Fall unseres Beispiels kommt uns zugute, dass wir a) nur in zwei Richtungen sortieren (aufsteigend und absteigend), und dass b) die Imageliste des Headers auch nur zwei Bitmaps enthält. Und weil die Variable "`SortOrder`" nur die Werte Null und Eins annehmen kann, können wir sie einfach als Index für das gewünschte Bild angeben:

```
    hi.iImage := SortOrder;
  end;
```

Mit "`Header_SetItem`" übergeben wir dem Header-Control die geänderten Einstellungen, und -voilà- erscheint der Pfeil im Spaltenkopf der List-View:



3.8. Der Tree-View

3.8.1. Den Tree-View erzeugen

Der Tree-View dürfte Ihnen von der VCL-Programmierung und diversen Programmen bekannt sein. Typischerweise wird er benutzt, um Informationen verschiedener Art baumartig anzuordnen. Der Windows Explorer ist ein solches Beispiel.

Das Erzeugen des Tree-View geht, wie bekannt, mit "`CreateWindowEx`" vonstatten, wobei wir den Klassennamen `WC_TREEVIEW` benutzen:

```
hTreeview  := CreateWindowEx(WS_EX_CLIENTEDGE,WC_TREEVIEW,nil,
  WS_VISIBLE or WS_CHILD or TVS_HASLINES or TVS_LINESATROOT
  or TVS_HASBUTTONS,0,0,10,10,wnd,IDC_TREEVIEW,hInstance,nil);
```

Stilattribute des Tree-View

Die Besonderheit in obigem Codeauszug sind die Attribute, die mit den normalen Fensterstilen angegeben werden können. Ich habe für dieses Beispiel die folgenden Flags benutzt und so einen typischen Tree-View erzeugt:

Wert	Bedeutung
TVS_HASLINES	der Tree-View zeigt die typischen Linien zwischen den Elementen
TVS_LINESATROOT	zeigt die Linien auch für die obersten Knoten der Hierarchie; TVS_HASLINES muss aber gesetzt sein, sonst wird das Flag ignoriert
TVS_HASBUTTONS	der Tree-View besitzt die bekannten Plus/Minus-Schaltflächen zum Ein- und Ausblenden von Knoten

Weitere, zum Teil bekannte, Stile sind

Wert	Bedeutung
TVS_CHECKBOXES	jedes Item besitzt eine Checkbox
TVS_EDITLABELS	der "in place"-Editor kann zum Ändern der Itembezeichnungen verwendet werden
TVS_SINGLEEXPAND	ein Klick genügt, um einen Knoten zu öffnen; ein evtl. anderer offener Knoten wird geschlossen (es sei denn, der Anwender hält die STRG-Taste gedrückt)
TVS_TRACKSELECT	aktiviert das "Hot tracking", ein Item wird bereits durch "Kontakt" mit dem Mauszeiger markiert

Bleiben wir für den Augenblick dabei und schauen uns eine kleine Zusatzfunktion des Beispielprogramms an: es ist möglich, ein paar Stile zur Laufzeit zu ändern. So lassen sich die Linien und Buttons sowie das "Hot tracking" ab- und anschalten.
Das Programm benutzt dazu eine kleine Funktion, die den Stil des Tree-View verändert und das Ergebnis (zwecks Kennzeichnung im Hauptmenü) zurückliefert:

```
function SetStyle(hTV: HWND; dwNewStyle: dword): dword;
var
  dwStyle : dword;
begin
  dwStyle := GetWindowLong(hTV,GWL_STYLE);

  if(dwStyle and dwNewStyle = 0) then
    SetWindowLong(hTV,GWL_STYLE,dwStyle or dwNewStyle)
  else
    SetWindowLong(hTV,GWL_STYLE,dwStyle and not dwNewStyle);

  Result := GetWindowLong(hTV,GWL_STYLE);
end;
```

Eine ähnliche Funktion kennen wir schon von der List-View und dem Umschalten der Ansicht. Im Fall des Tree-View, übergeben wir der Funktion das Handle desselben und das gewünschte Flag, das wir ein- oder ausschalten wollen. Nehmen wir als Beispiel das "Hot tracking", das sich im Beispielprogramm gleich aus zwei Flags zusammensetzt:

```
SetStyle(hTreeview,TVS_TRACKSELECT or TVS_SINGLEEXPAND);
```

Würden Sie die Funktion ein weiteres Mal aufrufen, würden die Flags wieder entfernt werden, und das "Hot tracking" wäre deaktiviert.

3.8.2. Den Tree-View füllen

Nach dem Erzeugen des Tree-View wollen wir diesen nun mit Inhalt füllen. Das Beispielprogramm lädt zu dem Zweck zuerst alle vorhandenen Festplatten bzw. Partitionen, begonnen bei C, und stellt diese im Hauptmenü dar. Die erste Partition wird dann auch gleich nach Ordnern durchsucht; diese Ordner sollen im Tree-View dargestellt werden - eine Art Mini-Explorer also ...

Das Scannen der Partition soll nur am Rande besprochen werden. Einzig erwähnenswert ist eine Eigenart, die Sie aber vielleicht schon kennen: Das allererste Scannen einer Festplatte oder Partition kann u.U. eine geraume Zeit in Anspruch nehmen. Aber bereits der zweite Start geht zügiger vonstatten. Dabei handelt es sich keineswegs um ein subjektives Empfinden; mit Hilfe von "`GetTickCount`" kann der Zeitunterschied sichtbar gemacht werden.

Doch zurück zum Thema: Um einen Eintrag zu erstellen, benötigen wir die Nachricht "`TVM_INSERTITEM`", die einen Zeiger auf das `TVInsertStruct`-Record erwartet. Uns interessiert von diesem Record fürs Erste der Parent, der immer auf den jeweils zuvor erzeugten Eintrag zeigen sollte. Beim ersten Aufruf der Prozedur "`Scan`", die für das Scannen der Partition zuständig ist, wird hier der Wert **nil** als "`hRoot`" übergeben:

```
tvi.hParent      := hRoot;
```

Um eine Explorer-ähnliche Sortierung zu erhalten, verwenden wir `TVI_SORT` als Wert für die Einfügeoperation:

```
tvi.hInsertAfter := TVI_SORT;
```

Weil wir im Augenblick nur blanke Texteinträge erstellen, reicht folgendes Flag:

```
tvi.item.mask    := TVIF_TEXT;
```

Und natürlich müssen wir dann auch den jeweils aktuellen Ordnernamen angeben, der von den Funktionen "`FindFirstFile`" und "`FindNextFile`" geliefert wird:

```
tvi.item.pszText := ds.cFileName;
```

Das war´s. Den so erstellten Eintrag fügen wir nun unserem Tree-View hinzu, indem wir "`SendMessage`" aufrufen und das Record als `lParam` übergeben. Das Ergebnis des Aufrufs ist dann das Handle des Eintrags:

```
hr               := HTREEITEM(SendMessage(hTV,TVM_INSERTITEM,0,LPARAM(@tvi)));
```

Und weil sich die Prozedur "`Scan`" im Beispielprogramm bei jedem gefundenen Ordner immer wieder selbst aufruft, müssen wir dieses Handle als neuen Root übergeben. Andernfalls sieht unser Baum nicht so aus wie wir das vom Explorer gewohnt sind.

Hinweis

Neben der Nachricht gibt es auch noch ein Makro, "`TreeView_InsertItem`", das wir stattdessen benutzen können. Es kapselt die o.g. Nachricht einfach nur, liefert aber auch gleich den Typ `HTREEITEM` zurück, so dass das o.g. Typecasting entfallen kann:

```
hr := TreeView_InsertItem(hTV,tvi);
```

Ich werde noch häufiger auf solche Funktionen eingehen, da ich sie in den meisten Fällen für den eleganteren und einfacheren Weg halte.

TVInsertStruct-Definition

```
typedef struct tagTVINSERTSTRUCT {
    HTREEITEM hParent;       // Handle des Parent-Items
    HTREEITEM hInsertAfter;  // Handle des Items, nach dem das
                             // aktuelle eingefügt werden soll;
                             // bzw. eine von vier vordefinierten
                             // Konstanten
#if (_WIN32_IE >= 0x0400)
    union
    {
        TVITEMEX itemex;     // itemex-Record zum aktuellen Item
        TVITEM item;         // item-Record zum aktuellen Item
    } DUMMYUNIONNAME;
#else
    TVITEM item;             // item-Record zum aktuellen Item
#endif
}
```

TVM_INSERT-Definition

```
TVM_INSERT
    wParam = 0;
    lParam = (LPARAM) (LPTVINSERTSTRUCT) lpis;
```

Vordefinierte Werte zum Einfügen von Items

Wert	Bedeutung
TVI_FIRST	Das Item wird am Anfang der aktuellen Ebene eingefügt.
TVI_LAST	Das Item wird am Ende der aktuellen Ebene angehangen.
TVI_ROOT	Das Item wird als Root der aktuellen Ebene eingefügt.
TVI_SORT	Das Item wird in alfabetischer Reihenfolge in die aktuelle Ebene eingefügt.

3.8.3. Symbole für den Baum

So ganz ohne Images wirkt unser Tree-View aber nicht. Oder? Also, dann wollen wir mal ... Wir hätten nun die Möglichkeit, das einfache Prinzip der List-View zu wiederholen, da wir auch hier eine ImageList brauchen. Die Grundlagen (sprich: Liste erzeugen, Icons laden und zuweisen) sind nämlich identisch. Aber auch hier wollen wir das System zu Hilfe nehmen und die Symbole verwenden, die Windows für Ordner benutzt. Es entspricht also dem erweiterten Prinzip der List-View. Vorteil: das Beispielprogramm benutzt, abhängig vom laufenden Betriebssystem, die gewohnten Symbole aus dem Explorer.

Fangen wir mit der Imageliste an. Das System stellt uns das Handle dieser Liste zur Verfügung, wenn wir die Funktion "`SHGetFileInfo`" mit dem Flag `SHGFI_SYSICONINDEX` aufrufen. Da uns außerdem nur die kleinen Symbole interessieren, geben wir das Attribut `SHGFI_SMALLICON` gleich mit an:

```
hSmallImg := HIMAGELIST(SHGetFileInfo('',0,fi,sizeof(fi),
  SHGFI_SYSICONINDEX or SHGFI_SMALLICON));
```

Der Rückgabewert der Funktion ist das Handle der Imageliste, die wir an den Tree-View weiterreichen:

```
if(hSmallImg <> 0) then
  TreeView_SetImageList(hTreeview,hSmallImg,TVSIL_NORMAL);
```

So weit, so gut.

Beim Einlesen der einzelnen Ordner benutzen wir nun die Funktion ein weiteres Mal und holen uns den Index der jeweiligen Symbole. Das hat damit zu tun, dass nicht alle Ordner das selbe Bild benutzen. Denken Sie an Ihre Favoriten, an den Verlauf, an die Eigenen Dateien, usw.

Wir ermitteln daher zuerst das normale Ordnersymbol. Der Aufruf ähnelt dem obigen, nur interessiert uns hier die `iIcon`-Membervariable des `TSHFileInfo`-Records. Diese `integer`-Variable enthält den Index des Symbols:

```
ZeroMemory(@fi,sizeof(TSHFileInfo));
SHGetFileInfo(ds.cFileName,0,fi,sizeof(fi),SHGFI_SMALLICON or
  SHGFI_SYSICONINDEX);
ico1 := fi.iIcon;
```

Für das Symbol des aufgeklappten Ordners (das wir sehen wollen, wenn wir in unserem Beispielprogramm einen Eintrag auswählen, benutzen wir den selben Aufruf, geben aber zusätzlich noch das Flag `SHGFI_OPENICON` an:

```
ZeroMemory(@fi,sizeof(TSHFileInfo));
SHGetFileInfo(ds.cFileName,0,fi,sizeof(fi),SHGFI_SMALLICON or
  SHGFI_OPENICON or SHGFI_SYSICONINDEX);
ico2 := fi.iIcon;
```

3.8.3.1. Neue Flags braucht der Tree-View ...

Von unseren Bildchen haben wir allerdings nichts, solange wir nicht den Code beim Scannen der Partitionen entsprechend ergänzen. Schauen wir uns die Sache noch einmal an. So sieht es bisher aus:

```
tvi.item.mask    := TVIF_TEXT;
tvi.item.pszText := ds.cFileName;
```

Zuerst erweitern wir die Flags, so dass auch die Symbole berücksichtigt werden. Da wir das normale und das geöffnete Ordnersymbol haben, benötigen wir auch die Flags für das normale und selektierte Image:

```
tvi.item.mask    := TVIF_TEXT or TVIF_IMAGE or TVIF_SELECTEDIMAGE;
```

Und dann fehlt eigentlich nur noch die Referenz auf die beiden Symbole, bzw. auf deren Indizes, wobei das Flag `TVIF_IMAGE` für die Membervariable `iImage` und `TVIF_SELECTEDIMAGE` für `iSelectedImage` zuständig ist:

```
tvi.item.iImage         := ico1;
tvi.item.iSelectedImage := ico2;
```

Voilà, das war´s. Unser Tree-View "erstrahlt" mit den Ordnersymbolen des Systems.

SHGetFileInfo-Definition

```
DWORD_PTR SHGetFileInfo(
    LPCTSTR pszPath,          // Datei- oder Ordnername, usw.
    DWORD dwFileAttributes,   // Attribute
    SHFILEINFO* psfi,         // TSHFileInfo-Variable
    UINT cbFileInfo,          // Größe der Variablen
    UINT uFlags               // Flags
);
```

TVM_SETIMAGELIST-DefinitionTVM_SETIMAGELIST

```
    wParam = (int) iImage;          // TVSIL_NORMAL = normale ImageList
                                    // TVSIL_STATE  = Status-ImageList
    lParam = (HIMAGELIST) himl;     // ImageList
```

3.8.4. Welcher Eintrag ist ausgewählt?

Es gibt eine relativ einfache Möglichkeit, den gerade markierten Eintrag herauszufinden: Sie reagieren in Ihrem Programm mindestens auf die Benachrichtigung "`TVN_SELCHANGED`", die als Teil von "`WM_NOTIFY`" gesendet wird.

"Mindestens" bedeutet, dass Sie alternativ auch auf "TVN_SELCHANGING" reagieren können. Diese Benachrichtigung wird ausgelöst, sobald sich die aktuelle Auswahl ändert. Dagegen wird "TVN_SELCHANGED" erst ausgelöst, sobald sich die Auswahl geändert hat.

Da im lParam der Zeiger auf ein TNMTreeView-Record übergeben wird, haben Sie so Zugriff auf das alte (vorher aktive) und das neue (auszuwählende) Item im Treeview, so dass Sie beides relativ einfach gegenüber stellen können. Zuerst erweitern Sie die mask-Membervariable des alten Items um das Flag TVIF_TEXT:

```
itemOld.mask       := TVIF_TEXT;
```

Als nächstes weisen Sie einen Textpuffer und dessen Größe zu. Wenn Sie dies nicht tun, dann sehen Sie entweder irgendwelche unsinnigen Zeichen oder bestenfalls gar nichts:

```
itemOld.pszText    := buf;
itemOld.cchTextMax := sizeof(buf);
```

Ob Sie danach die Membervariable pszText oder den Textpuffer nutzen, das bleibt Ihnen überlassen. Wichtig ist nur, dass Sie überhaupt einen Textpuffer angeben!

Mit Hilfe des Makros "TreeView_GetItem" können Sie dann den Text auslesen und an eine string-Variable übergeben:

```
if(TreeView_GetItem(hdr.hwndFrom,itemOld)) then
  s := string(itemOld.pszText);
```

Dieses Makro dient natürlich hauptsächlich dem Zweck, alle möglichen Informationen zu einem Item herauszufinden. Der Text ist da nur eine Sache. Letztlich entscheiden die benutzten Flags in der mask-Membervariablen, welche Informationen Sie interessieren.
Wie dem auch sei; die selben Anweisungen helfen uns auch, den Text des neuen Items (itemNew) herauszufinden. Wenn wir die string-Variable entsprechend erweitert haben, können wir sie in der Statuszeile anzeigen und sehen so, woher wir kamen und wohin wir gingen:

```
if(TreeView_GetItem(hdr.hwndFrom,itemNew)) then
  s := s + ' -> ' + string(itemNew.pszText);
SendMessage(hStatusBar,SB_SETTEXT,SB_SIMPLEID,LPARAM(pchar(s)));
```

Gehen wir noch einen Schritt weiter: Wir wollen in der Statuszeile nicht nur den Namen des aktuellen Items anzeigen, sondern wir wollen den kompletten Pfad bis hoch zum Root-Item sehen. Dazu übergeben wir das Handle des neuen Items zunächst an eine Variable vom Typ HTREEVIEW. Dies ist notwendig, da wir später eine **while**-Schleife benutzen werden, die solange durchlaufen wird bis wir das oberste Item erreicht haben.
Als verantwortungsbewusste und Ressourcen sparende Programmierer verwenden wir hier eine Variable, die uns beim Drag&Drop noch einmal begegnen wird:

```
HitHandle := itemNew.hItem;
```

Damit treten wir in die **while**-Schleife ein und füllen eine TTVItem-Variable, wie wir das am Anfang bereits getan haben:

```
while(HitHandle <> nil) do begin
  tv.mask       := TVIF_TEXT;
  tv.hItem      := HitHandle;
  tv.pszText    := buf;
  tv.cchTextMax := sizeof(buf);
```

Wenn wir den Text des aktuellen Items ausgelesen und an die string-Variable übergeben haben

```
  if(TreeView_GetItem(hdr.hwndFrom,tv)) then
    s := string(buf) + '\' + s
  else
    break;
```

dann holen wir uns mit Hilfe von "TreeView_GetParent" das Handle auf den jeweils übergeordneten Eintrag:

```
  HitHandle     := TreeView_GetParent(hdr.hwndFrom,HitHandle);
end;
```

Solange es ein solches übergeordnetes Item gibt, solange bleiben wir in der Schleife. Wird sie verlassen, weil es keine übergeordneten Items mehr gibt, oder weil ein Fehler auftrat, dann zeigt das Programm den kompletten Pfad in der Statuszeile an.

Im Beispielprogramm können Sie mit Hilfe einer Compilerdirektive (SHOWPATH) zwischen dieser Ansicht und der Gegenüberstellung von altem und neuem Item wechseln.

3.8.5. Ziehen und Fallen lassen

Im folgenden Kapitel werden wir uns mit dem Drag&Drop im Tree-View befassen. Das Verschieben und Kopieren der Einträge hat dabei natürlich keine Auswirkungen auf die Ordnerstruktur Ihrer Festplatte. Sie wissen ja inzwischen, dass das Beispielprogramm die Verzeichnisse Ihrer Festplatte(n) anzeigt.
Aber, wie eben gesagt, Sie müssen nicht befürchten, dass diese Ordner jetzt kopiert oder verschoben werden.

Wenn Sie Drag&Drop übrigens verhindern wollen, dann müssen Sie den Tree-View mit dem zusätzlichen Stilattribut TVS_DISABLEDRAGDROP erstellen.

3.8.5.1. Die Drag-Operation einleiten

Das Control sendet eine "TVN_BEGINDRAG"-Benachrichtigung (in Form von "WM_NOTIFY") an sein übergeordnetes Fenster, sobald der Anwender ein Item auswählt und es mit gedrückter linker Maustaste zu ziehen beginnt. Da Drag&Drop aber auch mit der rechten Maustaste möglich ist, lautet die Benachrichtigung evtl. auch "TVN_BEGINRDRAG". In beiden Fällen enthält der lParam-Parameter die Adresse auf ein TNMTreeView-Record, das alle Informationen zu dem ausgewählten Item enthält.

```
WM_NOTIFY:
  with PNMTreeView(lp)^ do
    case hdr.code of
      TVN_BEGINDRAG,
      TVN_BEGINRDRAG:
        begin
          { ... }
        end;
    end;
```

Unsere erste Aufgabe besteht darin, uns das aktuelle Item zu merken. Wir benutzen dazu eine globale HTREEITEM-Variable:

```
hOldItem := itemNew.hItem;
```

Als nächstes benötigen wir ein Bild, das während des Ziehens angezeigt werden soll. Sie kennen diesen Effekt sicher vom Windows Explorer: wenn Sie dort einen Ordner verschieben, dann sehen Sie transparent das Symbol und den Namen des Ordners, während im Hintergrund immer der Ordner markiert wird, über dem Sie sich gerade befinden. Würden wir das nicht tun, sehen wir beim Ziehen entweder gar nichts oder bestenfalls das Item, über dem wir uns gerade befinden (wenn wir diese Funktionalität denn einbauen).

Wir könnten nun spezielle Grafiken benutzen, wir können aber auch dem Tree-View-Control "sagen", dass es ein Bild erzeugen soll. Dabei hilft uns die Nachricht "TVM_CREATEDRAGIMAGE"

```
hDragImgList :=
SendMessage(hTreeview,TVM_CREATEDRAGIMAGE,0,LPARAM(itemNew.hItem));
```

bzw. folgendes Makro:

```
hDragImgList := TreeView_CreateDragImage(hTreeview,itemNew.hItem);
```

Das Handle des Items, das sozusagen als Bild benutzt werden soll, nehmen wir aus dem TNMTreeView-Record, dessen Adresse von "WM_NOTIFY" übermittelt wurde. (Nicht durcheinander kommen! TNMTreeView ist das Record, das hinter allem steht - weil wir aber nur die Adresse besitzen, brauchen wir PNMTreeView, um über den Zeiger auf das eigentliche Record zugreifen zu können.)

Der nächste Schritt sind die beiden Anweisungen "ImageList_BeginDrag" und "ImageList_DragEnter". Die Funktion "ImageList_BeginDrag" erwartet neben dem Index des Bildes auch dessen Hotspot. Dieser Punkt bezieht sich auf die linke obere Ecke des Bildes. Wir benutzen hier in beiden Fällen Null.

```
ImageList_BeginDrag(hDragImgList,0,0,0);
```

Würden Sie beispielsweise

```
ImageList_BeginDrag(hDragImgList,0,6,6);
```

verwenden, befände sich der Mauszeiger entsprechend weiter innerhalb des Bildes.

Zum Index ist zu sagen, dass unsere Liste nur ein Image enthält. Aus dem Grund muss natürlich in jedem Fall der Wert Null für den Index benutzt werden. Sie dürfen "Bild" dabei nicht ausschließlich auf das Ordnersymbol beziehen. Tatsächlich erzeugt das TreeView-Control ein Bild, das sowohl aus dem benutzten Symbol als auch aus dem Text des Items besteht. Würden Sie testweise einen anderen Index angeben, würden Sie alles mögliche sehen - aber nicht das Image, auf das es uns ankommt:

```
ImageList_BeginDrag(hDragImgList,1,0,0);
```

Die zweite Funktion "ImageList_DragEnter" verhindert, dass ein Fenster während des Ziehens verändert wird und zeigt das Bild an der angegebenen Position an. Da es um den Tree-View geht, übergeben wir dessen Fensterhandle. Und die Position stammt hier wieder aus dem TNMTreeView-Record, das durch "WM_NOTFIY" im lParam übermittelt wurde:

```
ImageList_DragEnter(hTreeview,ptDrag.x,ptDrag.y);
```

Als nächstes sorgen wir mit "SetCapture" dafür, dass alle Mausnachrichten direkt an unser Hauptfenster geleitet werden. Das macht auch Sinn, da man die Maus während einer Drag&Drop-Aktion schwerlich anderweitig nutzen kann. Zusätzlich nutzen wir noch eine bool-Variable zur Kontrolle, ob wir uns im "Drag-Modus" befinden:

```
SetCapture(wnd);
DragMode := true;
```

3.8.5.2. Hier kommt die Maus ...

Angesprochen habe ich es bereits: im Windows Explorer wird während des Ziehens immer automatisch der Ordner markiert, über dem sich der Mauszeiger gerade befindet. Und natürlich wird auch das transparente Bild des gezogenen Ordners entsprechend verschoben.
Um diesen Effekt zu erreichen, bemühen wir die Nachricht "WM_MOUSEMOVE, wobei Sie die auch schon erwähnte bool-Variable heranziehen. Das ist insbesondere dann notwendig, wenn Sie in Ihrem Programm sowieso auf Mausbewegungen reagieren, damit diese nicht als Drag-Operation missinterpretiert werden:

```
WM_MOUSEMOVE:
  if(DragMode) then begin
    { ... }
  end;
```

Zuerst bewegen wir das transparente Drag-Image mit Hilfe von "ImageList_DragMove" auf die aktuelle Position des Mauszeigers. Weil "WM_MOUSEMOVE" diese Position im lParam mitbringt, nutzen wir sie:

```
ImageList_DragMove(LOWORD(lp),HIWORD(lp));
```

Effekt #2: Wir markieren das Item im Tree-View, über dem wir uns gerade befinden. Dazu benötigen wir zuerst eine Variable vom Typ TTVHitTestInfo, der wir die Mausposition übergeben:

```
tvhit.pt.x := LOWORD(lp);
tvhit.pt.y := HIWORD(lp);
```

Mit Hilfe eines so genannten "hit test" (mit "*Kollisionsabfrage*" durchaus treffend übersetzt) können wir dann überprüfen, ob sich an der angegebenen Position ein Item befindet. Dazu bietet sich entweder die Nachricht "TVM_HITTEST" oder wieder ein Makro, "TreeView_HitTest", an. Wenn Sie die Nachricht verwenden wollen, müssen Sie die TTVHitTestInfo als lParam übergeben. Mit dem Makro ist es einfacher:

```
HitHandle  := TreeView_HitTest(hTreeview,tvhit);
```

Befindet sich nun ein Item "unter" der aktuellen Mausposition, dann entspricht der Rückgabewert dem Handle des Items. Das Makro liefert das Handle bereits als HTREEITEM zurück; das Ergebnis der Nachricht müssten Sie erst entsprechend casten.
Wenn wir nun also ein Item haben, dann markieren wir es mit der Nachricht "TVM_SELECTDROPTARGET" oder mit dem Makro:

```
if(HitHandle <> nil) then
  TreeView_SelectDropTarget(hTreeview,HitHandle);
```

Zur Sicherheit sollten Sie das Drag-Image zuvor verstecken und danach wieder sichtbar machen. Auf die Weise vermeiden Sie unschöne Fragmente im Tree-View

Zum Verstecken und Anzeigen benutzen Sie "ImageList_DragShowNolock". Der bool-Parameter entscheidet dabei, ob das Bild versteckt (**false**) oder angezeigt (**true**) wird. Im Beispielprogramm finden Sie die Anwendung dieser Funktion.

3.8.5.3. Man muss auch loslassen können ...

Letzter Schritt beim Drag&Drop ist, natürlich!, das Loslassen. Unser Programm muss daher die Nachricht "WM_LBUTTONUP" (linke Maustaste) und/oder "WM_RBUTTONUP" (rechte Maustaste) bearbeiten. Wenn in beiden Fällen das selbe passieren soll, können Sie die Bearbeitung natürlich entsprechend zusammenfassen. Wir gehen erst einmal vom Normalfall aus; sprich: wir haben die linke Maustaste benutzt.

Zuerst geben wir das Tree-View-Fenster wieder frei, damit es aktualisiert werden kann. Dazu dient die Funktion "ImageList_DragLeave", die gleichzeitig auch das Drag-Bild verschwinden lässt.

```
ImageList_DragLeave(hTreeview);
```

Als Zeichen, dass die Drag-Operation beendet ist, rufen wir dann "ImageList_DragEnd" auf:

```
ImageList_EndDrag;
```

Und da wir die Imagelist nicht mehr benötigen, "zerstören" wir sie

```
ImageList_Destroy(hDragImgList);
```

Danach sorgen wir dafür, dass die Mausnachrichten wieder wie gewohnt bearbeitet werden können. Da wir das Item ja losgelassen haben, ist es auch nicht mehr erforderlich, die Mausnachrichten auf unser Fenster zu begrenzen. Die `bool`-Variable, die uns zur Kontrolle diente, wird bei der Gelegenheit auch wieder zurückgesetzt:

```
ReleaseCapture;
DragMode := false;
```

Zu guter Letzt markieren wir das Item, das sich zuletzt unter der Position des Mauszeigers befand. Dazu benutzen wir zwei Funktionen: "`TreeView_SelectItem`" und "`TreeView_GetDropHilite`". Zur letzteren ist zu sagen, dass sie in Microsofts PSDK eigentlich den Namen "`TreeView_GetDropHilight`" trägt.
Borland hat ihr aber (zumindest in Delphi 5) einen etwas anderen Namen gegeben. Ob andere Delphi-Versionen den korrekten Namen benutzen, kann ich leider nicht sagen. Im Zweifelsfall sollten Sie es darauf ankommen lassen. Der Compiler wird sich schon melden; so oder so ...

Was tun die beiden Funktionen nun? - "`TreeView_SelectItem`" markiert einen Eintrag im Control und erwartet dessen Handle. Dieses Handle erhalten wir u.a. mit "`TreeView_GetDropHilite`", wobei uns diese Funktion den Eintrag liefert, der sich zuletzt unter dem Mauszeiger befand. Und das wollten wir ja auch:

```
TreeView_SelectItem(hTreeview,TreeView_GetDropHilite(hTree-View));
```

Wenn Sie lieber mit den Nachrichten des Controls arbeiten, dann können Sie auch "`TVM_SELECTITEM`" und "`TVM_GETNEXTITEM`" (in Verbindung mit dem `TVGN_DROPHILITE`-Flag) verwenden:

```
SendMessage(hTreeview,TVM_SELECTITEM,TVGN_CARET,
  SendMessage(hTreeview,TVM_GETNEXTITEM,TVGN_DROPHILITE,0));
```

3.8.6. Einträge kopieren und verschieben

Zum Kopieren von Einträgen können wir auf Bewährtes zurückgreifen. Wir haben bereits besprochen, wie man den Text des markierten Items herausfinden kann.
Das Handle des gezogenen Items haben wir bereits gesichert, also weisen wir es wieder einer `TTVItem`-Variablen zu:

```
tv.hItem      := itemFrom;
```

Diesmal interessiert uns aber mehr als nur der Text des Items, also sieht unsere Flag-Kombination entsprechend aus:

```
tv.mask       := TVIF_HANDLE or TVIF_TEXT or TVIF_CHILDREN or
  TVIF_IMAGE or TVIF_SELECTEDIMAGE or TVIF_PARAM;
```

Hier interessieren uns neben Text auch die Images, die der Eintrag benutzt, sowie ein `lParam` (falls verwendet). Und entscheidend für das Kopieren des Items ist auch die Frage, ob dieser untergeordnete Einträge hat oder nicht.

Das Auslesen des Items funktioniert dann so, wie es bereits beschrieben wurde. Für den Text benötigen wir einen Puffer und dessen Größe. Dann holen wir die Informationen des Items mit Hilfe von "`TreeView_GetItem`".
Wenn wir den Eintrag haben, können wir ihn an die neue Position kopieren. Die neue Position ist dabei natürlich der Eintrag, der sich beim Abschluss der Drag-Operation unter dem Mauszeiger befunden hat.
Zum Kopieren (oder besser: Einfügen) benötigen wir wieder das schon bekannte Record vom Typ `TTVInsertStruct`, das wie folgt gefüllt wird:

```
tvi.hParent      := itemTo;
tvi.hInsertAfter := TVI_SORT;
tvi.item         := tv;
```

Dann rufen wir "TreeView_InsertItem" auf, übergeben dieses Record und erhalten als Ergebnis das Handle des neu eingefügten Items:

```
parent           := TreeView_InsertItem(hTV,tvi);
```

Jetzt ist die Frage: gibt es weitere, untergeordnete Items? Die Antwort gibt uns die Membervariable cChildren des TVItem-Records. Leider liefert uns diese Variable lediglich die Werte Null (es gibt keine untergeordneten Einträge), Eins (es gibt einen oder mehrere untergeordnete Einträge) und I_CHILDRENCALLBACK als Ergebnis zurück. Die Konstante und den Wert Null können wir in unserem Beispiel getrost ignorieren. Uns interessiert nur der Wert Eins.

Also holen wir uns den ersten untergeordneten Eintrag mit Hilfe von "TreeView_GetChild":

```
child := TreeView_GetChild(hTV,itemFrom);
```

Weil unser Beispielprogramm eine eigene Prozedur zum Kopieren der Einträge benutzt, brauchen wir diese nur aufzurufen, wobei wir die neu ermittelten Werte für das zu kopierende und das zu empfangende Item angeben:

```
CopyItems(hTV,child,parent);
```

Da es aber mehrere Einträge geben kann, benutzen wir nun "TreeView_GetNextSibling". Diese Funktion liefert das jeweils nächsten Item, das sich auf der selben Ebene befindet. Weil wir uns nun aber auf der Ebene des ersten Child-Items befinden, bekommen wir auf die Weise alle untergeordneten Einträge:

```
child := TreeView_GetNextSibling(hTV,child);
```

Das Beispielprogramm verwendet auch hier wieder eine **while**-Schleife, die solange durchlaufen wird bis "child" **nil** ist. Und weil sich die Prozedur "CopyItems" auf diese Weise immer wieder selbst aufruft, werden alle untergeordneten und unter-unter-unter-...-geordneten Einträge kopiert.

Unsere letzte Aufgabe ist es daher nur noch, den Originaleintrag zu entfernen - sofern wir das wollen. Im Beispielprogramm geschieht dies, wenn Sie einen Eintrag mit der linken Maustaste verschieben, bzw. wenn Sie die rechte Maustaste benutzen und aus dem Popupmenü dann natürlich "Eintrag verschieben" auswählen. Die Funktionsweise ist dabei recht einfach: Sie rufen "TreeView_DeleteItem" auf und übergeben das Handle des, vor der Drag-Operation gesicherten Items:

```
TreeView_DeleteItem(hTreeview,hOldItem);
```

3.8.7. Der "in place"-Editor

Auch der Tree-View besitzt einen eingebauten Editor, mit dem Sie den angezeigten Text eines Items ändern können. Für die Grundlagen möchte ich Sie daher an den entsprechenden Beitrag der List-View verweisen, weil das Prinzip das gleiche ist.
Einzig die gesendeten Benachrichtigungen sind, natürlich!, bei einem Tree-View anders. Streng genommen unterscheiden sie sich aber auch nur vom T am Anfang - "TVN_BEGINLABELEDIT" anstelle von "LVN_BEGINLABELEDIT" z.B.

Ich will es daher mit einem kurzen Codeausriss bewenden lassen, der zeigt, wie (nach dem Ändern der Itembeschriftung) alter und neuer Text gegenüber gestellt werden. Das Prinzip entspricht dabei der List-View-Demonstration:

```
TVN_ENDLABELEDIT:
  if(PTVDispInfo(lp)^.item.pszText <> '') then begin
    tv.hItem      := PTVDispInfo(lp)^.item.hItem;
    tv.mask       := TVIF_TEXT;
    tv.pszText    := buf;
    tv.cchTextMax := sizeof(buf);

    if(TreeView_GetItem(hdr.hwndFrom,tv)) then begin
      MessageBox(wnd,pchar(Format('"%s" vs. "%s"',
        [buf,PTVDispInfo(lp)^.item.pszText])),nil,0);

      Result := 1;
    end;
  end;
```

Und wie bei der List-View können Sie auch hier einen Shortcut definieren, der den Editmodus auslöst. Shortcut ist vielleicht nicht der passende Begriff, da Ihnen eigentlich der Tree-View mit der Benachrichtigung "TVN_KEYDOWN" die Abfrage von bestimmten Tasten erlaubt.
Ich habe mich wieder für F2 entschieden, da dies ja auch aus dem Windows Explorer bekannt ist:

```
TVN_KEYDOWN:
  if(PTVKeyDown(lp)^.wVKey = VK_F2) then begin
    HitHandle := TreeView_GetSelection(hdr.hwndFrom);
    if(HitHandle <> nil) then begin
      SetFocus(hdr.hwndFrom);
      TreeView_EditLabel(hdr.hwndFrom,HitHandle);
    end;
  end;
```

Ich sollte aber erwähnen, dass in Delphi 5 die Deklaration von PTVKeyDown fehlt. Die Unit "CommCtrl.pas" enthält zwar das Record (tagTVKEYDOWN) und die dazu passende Borland-typische Deklaration (TTVKeyDown), aber die Zeigerdefinition fehlt. Ich habe daher den Typ im Beispielprogramm selbst deklariert:

```
type
  PTVKeyDown = ^TTVKeyDown;
```

Möglicherweise ist das aber in Ihrer Delphi-Version nicht mehr erforderlich.

3.9. Die Rebar

3.9.1. Das Rebar-Control erzeugen

Das Rebar-Control ist Ihnen möglicherweise unter dem Namen Coolbar (Delphi-VCL) bekannt. Es handelt sich dabei um ein Control, das seinerseits andere Controls kapseln kann. Sie können beispielsweise Toolbars, Buttons, Comboboxen, usw. als so genannte "Bands" einfügen. So stellt beispielsweise der Internet Explorer seine Toolbars dar.
Für dieses Kapitel wollen wir grob auf das Beispiel aus Microsofts PSDK zurückgreifen.

Das Erzeugen des Rebar-Controls unterscheidet sich nicht weiter von den bisher besprochenen Controls. Sie benutzen wie üblich den Befehl "CreateWindowEx" mit dem Klassennamen des Controls (hier REBARCLASSNAME):

```
hwndRebar := CreateWindowEx(WS_EX_TOOLWINDOW,REBARCLASSNAME,nil,
  WS_VISIBLE or WS_BORDER or WS_CHILD or WS_CLIPCHILDREN or
  WS_CLIPSIBLINGS or RBS_VARHEIGHT or RBS_BANDBORDERS,0,0,0,0,
  hwndParent,IDC_REBAR,hInstance,nil);
```

InitCommonControlsEx

Wenn Sie den Befehl "InitCommonControlsEx" verwenden, müssen Sie für die dwICC-Membervariable die Klasse ICC_COOL_CLASSES benutzen.

3.9.2. Spezielle Fensterstile

Schauen wir uns kurz die benutzten Stilattribute und weitere Möglichkeiten an:

Wert	Bedeutung
RBS_VARHEIGHT	Die jeweiligen "Bands" können unterschiedliche Höhen haben. Ohne dieses Attribut wird für alle "Bands" die gleiche Höhe benutzt.
RBS_BANDBORDERS	Die einzelnen "Bands" sind optisch voneinander getrennt.
RBS_VERTICALGRIPPER	Wird das Rebar-Control mit dem Stil CCS_VERT erstellt, wird der Gripper (das Element zum Vergrößern und Verkleinern des Bandes) vertikal dargestellt.
RBS_AUTOSIZE	Das Control passt automatisch die Bänder an, wenn sich seine Größe oder Position ändert.

(Weitere Attribute finden Sie wie immer im MSDN oder PSDK von Microsoft.)

3.9.3. Bänder erzeugen

Für unser Beispiel wollen als untergeordnete Elemente eine Combobox und einen Button erzeugen, die wir dann in das Rebar-Control einfügen. Zum Erzeugen dieser Elemente ist eigentlich nichts weiter zu sagen. Ich empfehle Ihnen im Zweifelsfall einen Blick in die entsprechenden Beiträge. Hier soll als Beispiel lediglich die Combobox gezeigt werden:

```
hwndChild := CreateWindowEx(0,'combobox',nil,WS_VISIBLE or WS_CHILD or
  WS_TABSTOP or WS_VSCROLL or WS_CLIPCHILDREN or WS_CLIPSIBLINGS or
  CBS_AUTOHSCROLL or CBS_DROPDOWN,0,0,100,200,hwndRebar,IDC_COMBOBOX,
  hInstance,nil);
```

Wichtig ist nur, wie Sie an der roten Hervorhebung sehen können, dass Sie das Element als Child des Rebar-Controls erzeugen.

3.9.3.1. Was ist nun ein Band?

Unter einem Band versteht man den Bereich, den ein Control innerhalb der Rebar belegt, etwa:

Mit dem so genannten "Gripper" (die Hervorhebung auf der linken Seite) lässt sich das Band vergrößern oder verkleinern. Ein Rebar-Control kann mehrere solcher Bänder enthalten. Auf diese Weise kann der Anwender das vorgegebene Aussehen an seine eigenen Wünsche anpassen, usw.

Um nun die Combobox als Band in die Rebar einzufügen, benötigen wir zunächst ihre Abmessungen, da wir die entsprechenden Stilattribute nutzen wollen:

```
GetWindowRect(hwndChild,rc);
```

Als nächstes benutzen wir eine `TRebarBandInfo`-Variable, die wir mit ihrer Größe initialisieren müssen:

```
ZeroMemory(@rbbi,sizeof(TRebarBandInfo));
rbbi.cbSize     := sizeof(TRebarBandInfo);
```

In der `fMask`-Membervariablen geben wir die gewünschten Flags an, die uns interessieren:

```
rbbi.fMask      := RBBIM_SIZE or RBBIM_CHILD or RBBIM_CHILDSIZE or
  RBBIM_ID or RBBIM_STYLE or RBBIM_TEXT;
```

Wert	Bedeutung
RBBIM_SIZE	Die `cx`-Membervariable ist gültig.
RBBIM_CHILD	Die `hwndChild`-Membervariable ist gültig.Toolbar-Bitmap-Konstanten1
RBBIM_CHILDSIZE	`cxMinChild`, `cyMinChild`, `cyChild`, `cyMaxChild` und `cyIntegral` sind gültig.
RBBIM_ID	`wId` ist gültig.
RBBIM_STYLE	Die `fStyle`-Membervariable ist gültig.
RBBIM_TEXT	Die Membervariable `lpText` ist gültig.

(weitere Flags finden Sie im MSDN oder PSDK)

Alle diese Flags haben Einfluss auf die Gestaltung des Bandes. Nehmen wir zunächst die Abmessungen, die durch `RBBIM_SIZE` und `RBBOM_CHILDSIZE` definiert werden. So wird hier festgelegt, dass die minimalen Abmessungen des Bandes nicht kleiner als die Höhe und Breite der Combobox sein dürfen, die während des Erstellens gültig waren:

```
rbbi.cxMinChild := rc.Right - rc.Left;
rbbi.cyMinChild := rc.Bottom - rc.Top;
```

Würden Sie beispielsweise den Wert von `cxMinChild` auf Null setzen, könnte das Band sozusagen bis zum Anschlag verkleinert werden, und das enthaltene Control wäre nicht mehr zu sehen. - Doch zurück zum Thema: als Anfangsbreite des Bandes legen wir 100 Pixel fest:

```
rbbi.cx         := 100;
```

Zu beachten ist, dass die Größe des Bandes aber auch von der Fensterbreite abhängig ist. Es kann also passieren, dass das Band viel breiter ist als Sie hier angeben. Das hängt auch davon ab, ob das nachfolgende Band in einer neuen Zeile begonnen werden soll oder nicht.

Als nächstes wären da die beiden Stilattribute:

```
rbbi.fStyle      := RBBS_CHILDEDGE or RBBS_GRIPPERALWAYS;
```

Wert	Bedeutung
RBBS_CHILDEDGE	das Band hat am eine Kante am oberen und unteren Rand.
RBBS_GRIPPERALWAYS	Der Gripper wird angezeigt, auch wenn nur ein Band im Rebar-Control vorhanden ist.
RBBS_BREAK	Das Band wird in einer neuen Zeile begonnen.

(s. MSDN oder PSDK für weitere Stilattribute)

Gemäß unserer Flag-Auswahl müssen wir dem Band noch eine eindeutige ID und das Handle der Combobox übergeben:

```
rbbi.wID         := IDC_COMBOBOX;
rbbi.hwndChild   := hwndChild;
```

Zu guter Letzt geben wir dem Band noch einen Namen, der (dank des Flags `RBBIM_TEXT`) auch angezeigt wird:

```
rbbi.lpText      := 'ComboBox';
```

Mit der Nachricht "`RB_INSERTBAND`" erzeugen wir nun das erste Band, wobei der `lparam` ein Zeiger auf die `TRebarBandInfo`-Variable ist. Der `wparam` enthält den Index, wo das Band eingefügt werden soll. Wird hier der Wert -1 benutzt, wird das neue Band am Ende (hinter dem jeweils letzten Band) angehangen:

```
SendMessage(hwndRebar,RB_INSERTBAND,WPARAM(-1),LPARAM(@rbbi));
```

3.9.4. Hintergrund-Bitmaps

Als Ergänzung wollen wir uns noch ansehen, wie man eine Bitmap als Hintergrund für ein Band nutzen kann. Vorausgesetzt wird, dass sich die gewünschte Bitmap in den Ressourcen des Programms befindet. Wenn dies der Fall ist, werden zunächst die Flags um eine erforderliche Angabe erweitert:

```
rbbi.fMask       := { ... } or RBBIM_BACKGROUND;
```

Damit können wir die Bitmap laden und an die `hbmBack`-Membervariable übergeben:

```
rbbi.hbmBack     := LoadBitmap(hInstance,MAKEINTRESOURCE(IDC_BACKBMP));
```

Rein prinzipiell war es das schon. Wenn Sie allerdings noch Wert auf eine Art "Wasserzeicheneffekt" legen (sprich: das Hintergrundbild soll nicht in irgendeiner Form verändert werden, sobald das Band verkleinert oder vergrößert wird), dann geben Sie am besten noch das Stilattribut `RBBS_FIXEDBMP` an:

```
rbbi.fStyle      := { .. } or RBBS_FIXEDBMP;
```

Das war es auch schon.

TRebarBandInfo-Definition

```
typedef struct tagREBARBANDINFO{
    UINT        cbSize;      // Recordgröße
    UINT        fMask;       // Flags
    UINT        fStyle;      // Stilattribute
    COLORREF    clrFore;     // Vordergrundfarbe
    COLORREF    clrBack;     // Hintergrundfarbe
    LPTSTR      lpText;      // Bandtext
    UINT        cch;
    int         iImage;      // Imageindex für Imagelisten
    HWND        hwndChild;   // Handle des Child-Controls
    UINT        cxMinChild;  // min. Breite des Child-Controls
    UINT        cyMinChild;  // min. Höhe des Child-Controls
    UINT        cx;          // Länge des Bandes in Pixeln
    HBITMAP     hbmBack;     // Hintergrundbitmap
    UINT        wID;         // ID des Bandes
#if (_WIN32_IE >= 0x0400)
    UINT        cyChild;
    UINT        cyMaxChild;
    UINT        cyIntegral;
    UINT        cxIdeal;
    LPARAM      lParam;
    UINT        cxHeader;
#endif
 } REBARBANDINFO
```

3.9.5. Eine ImageList mit Icons zuweisen

Das Rebar-Control erlaubt es, Icons auf einzelnen Bändern anzuzeigen. Dazu ist eine ImageList erforderlich, die Sie bereits von der List-View kennen. Etwas wirklich Neues gibt es daher auch nicht. Wir erzeugen zuerst die Liste für 32x32-Symbole:

```
himlRebar  := ImageList_Create(32,32,ILC_COLORDDB or ILC_MASK,1,0);
```

Bitte beachten Sie, dass das Beispielprogramm nur ein Symbol enthält. Es ist hier (in diesem ganz speziellen Fall) auch nicht möglich, weitere Symbole in die Liste einzufügen! Wie dem auch sei, das Symbol selbst holen wir wie gewohnt aus den Programmressourcen und reichen es an die ImageList weiter:

```
hIco       := LoadIcon(hInstance,MAKEINTRESOURCE(IDC_MAINICON));
ImageList_AddIcon(himlRebar,hIco);
```

Damit das Rebar-Control diese Liste nun auch verwenden kann, benötigen wir das `TRebarInfo`-Record, das wir zuerst initialisieren müssen:

```
ZeroMemory(@rbi,sizeof(rbi));
rbi.cbSize := sizeof(TRebarInfo);
```

Weil dieses Record bisher nur die Übergabe von Imagelisten an das Rebar-Control unterstützt, ist die Arbeit damit natürlich recht einfach. Es gibt nur ein einziges Flag, und selbstverständlich ist auch das Handle unserer Imageliste von Bedeutung:

```
rbi.fMask  := RBIM_IMAGELIST;
rbi.himl   := himlRebar;
```

Mit der Nachricht "`RB_SETBARINFO`" wird die Liste nun zu guter Letzt an das Control übergeben und steht uns somit dann auch dort zur Verfügung

```
SendMessage(hwndRebar,RB_SETBARINFO,0,LPARAM(@rbi));
```

3.9.5.1. Icons benutzen

Das reicht aber noch nicht ganz. Das Symbol muss erst noch einem Band zugeordnet werden bevor wir es sehen können. Dazu erweitern wir die Flags der `fMask`-Membervariablen um die folgende Angabe:

```
rbbi.fMask      := { ... } or RBBIM_IMAGE;
```

Sie besagt, dass die `iImage`-Membervariable gültig wird und den Index des Symbols aus der Imageliste enthalten muss. Da wir nur ein Symbol haben, ist dessen Index natürlich Null, was wir hiermit angeben:

```
rbbi.iImage     := 0;
```

Voilà, würden Sie das Band mit diesen erweiterten Angaben erzeugen, wäre vor dem Bandtext das Symbol zu sehen. Im Beispielprogramm können Sie dies mit Hilfe des Compilerschalters `USEBMPICO` nachvollziehen, der standardmäßig nicht verwendet wird.

TRebarInfo-Definition

```
typedef struct tagREBARINFO {
    UINT cbSize;      // Recordgröße
    UINT fMask;
    HIMAGELIST himl;  // Handle der ImageList
} REBARINFO
```

3.9.6. Bänder vergrößern und verkleinern

Zum Vergrößern und Verkleinern bestimmter Bänder stehen die beiden Nachrichten `RB_MAXIMIZEBAND` und `RB_MINIMIZEBAND` zur Verfügung. Der `wParam` ist in beiden Fällen den Index des gewünschten Bandes (null-basierend). Der `lParam`-Wert kann in beiden Fällen Null sein, aber nur bei der Nachricht `RB_MAXIMIZEBAND` ist auch ein Wert größer als Null möglich. Ein Wert größer Null wäre dann der Idealwert des Bandes, der zum Maximieren benutzt wird:

```
SendMessage(hRB,RB_MAXIMIZEBAND,0,300);
```

In dem Fall würde das Band auf eine Breite von genau 300 Pixel maximiert werden. Wird stattdessen Null benutzt, wird das Band so weit wie möglich maximiert.:

```
SendMessage(hRB,RB_MAXIMIZEBAND,0,0);
```

Wie gesagt: beim Minimieren eines Bandes kann `lParam` grundsätzlich nur Null sein:

```
SendMessage(hRB,RB_MINIMIZEBAND,1,0);
```

Hinweis

Nach dieser Methode ist das erste Band immer das, welches sich an erster Stelle im Rebar-Control befindet. Wenn Sie die Bänder verschieben, ändert sich dementsprechend auch ihr Index.
Wenn Sie aber das zuerst erzeugte Band minimieren oder maximieren wollen, unabhängig von seiner Position innerhalb der Rebar, dann müssen Sie seine ID in den aktuell gültigen Index umwandeln. Dazu verwenden Sie "`RB_IDTOINDEX`", wobei der `wparam` die entsprechende ID ist. In unserem Beispiel war das zuerst erzeugte Control die Combobox, die sich im Band mit der ID der Combobox (`IDC_COMBOBOX`) befindet. Den Index dieses Bandes erhalten wir daher so:

```
i := SendMessage(hRB,RB_IDTOINDEX,IDC_COMBOBOX,0);
```

Ist der Wert ungleich -1, können wir nun eine der o.g. Nachrichten zum Minimieren oder Maximieren des Bandes benutzen, wobei wir den ermittelten Indexwert als `wparam` angeben.

Benutzen zwei Bänder die selbe ID, wird der Index des zuerst gefundenen Bandes zurückgeliefert.

3.9.7. Chevrons

In diesem Kapitel wollen wir uns noch die so genannten "Chevrons" anschauen. Dabei handelt es sich um die kleinen Buttons mit dem Doppelpfeil am rechten Rand eines Bandes, die nur zu sehen sind, wenn das im Band enthaltene Control nicht vollständig dargestellt werden kann. Bekannt ist dieses Verhalten von Toolbars, Menüs u.ä. Elementen.

Allerdings setzt es min. die Version 5.80 der "comctl32.dll" voraus. Im Zweifelsfall hilft hier ein Update des Internet Explorer. Natürlich muss auch Ihre Delphi-Version aktuell genug sein und die neuen Konstanten, Nachrichten und Records kennen. Für Besitzer von Delphi 5 liegt eine spezielle Unit ("CommCtrl_Fragment.pas") bei, die diese Angaben enthält. Besitzer älterer Versionen können diese Unit testweise einsetzen, und Besitzer neuerer Versionen benötigen sie wahrscheinlich gar nicht mehr. Ich gehe also davon aus, dass Sie (auf die eine oder andere Weise) über die entsprechenden Deklarationen verfügen.

Aus dem Grund rüsten wir unser Beispielprogramm mit einer zusätzlichen Toolbar aus. Das bietet gleich einen Vorteil: ich kann und muss Sie auf eine Besonderheit hinweisen. Diese Besonderheit findet sich ein bisschen versteckt im PSDK:

PSDK:

The toolbar default sizing and positioning behaviors can be turned off by setting the CCS_NORESIZE and CCS_NOPARENTALIGN common control styles. Toolbar controls that are hosted by rebar controls **must** set these styles because the rebar control sizes and positions the toolbar.

Auf gut Deutsch: Toolbars, die sich im Band einer Rebar befinden, müssen die Stilattribute CCS_NORESIZE und CCS_NOPARENTALIGN benutzen, weil das Rebar-Control die Position und Größe der Toolbar kontrolliert. Wenn Sie diese Attribute vergessen oder weglassen, erhalten Sie alle möglichen Ergebnisse - nur wird die Toolbar nie im Rebar-Control erscheinen. Oder sagen wir: nicht fehlerfrei! Unser Aufruf sieht daher so aus:

```
hwndChild := CreateWindowEx(0,TOOLBARCLASSNAME,nil,WS_CHILD or
  WS_VISIBLE or CCS_NODIVIDER or CCS_NORESIZE or CCS_NOPARENTALIGN or
  TBSTYLE_FLAT,0,0,0,0,hwndRebar,IDC_TOOLBAR,hInstance,nil);
```

Als Parent wird wieder, wie gehabt, das Rebar-Control angegeben. Es macht aber auch keinen Unterschied wenn Sie direkt das Handle des Hauptfensters angeben ("hwndParent" im Beispielprogramm).
Das Erzeugen der Buttons und Bitmaps einer Toolbar soll hier ebenfalls nicht das Thema sein. Dafür möchte ich Sie auf das Toolbar-Tutorial verweisen. Wir widmen uns lieber den Chevrons -

Um den kleinen Button anzeigen zu können, müssen wir zuerst eine Idealgröße für das Band festlegen. Wird diese Idealgröße unterschritten, erscheint rechts der Chevron und weist den Benutzer darauf hin, dass einige Elemente verborgen sind. Um die Idealgröße aber überhaupt beim Erzeugen des Bandes angeben zu können, ist das zusätzliche Flag RBBIM_IDEALSIZE erforderlich:

```
rbbi.fMask      := { ... } or RBBIM_IDEALSIZE;
```

Damit steht uns die Membervariable cxIdeal zum Befüllen zur Verfügung. Ich habe die Idealgröße der Toolbar vorher in einer Schleife berechnen lassen, so dass die tatsächliche Breite jedes Buttons (auch von Separatoren, die ja naturgemäß schmaler sind) berücksichtigt wird:

```
iIdeal := 0;
for i  := 0 to SendMessage(hwndChild,TB_BUTTONCOUNT,0,0) - 1 do begin
  SendMessage(hwndChild,TB_GETITEMRECT,WPARAM(i),LPARAM(@rc));
  inc(iIdeal,(rc.Right - rc.Left));
end;
```

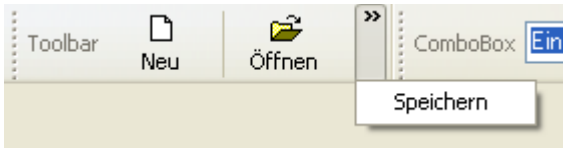
Diesen Wert übergeben wir nun an die Membervariable des Records:

```
rbbi.cxIdeal    := iIdeal;
```

Zu guter Letzt geben wir ein zusätzliches Stilattribut an, damit der Chevron auch tatsächlich erscheint:

```
rbbi.fStyle      := { ... } or RBBS_USECHEVRON;
```

Das Band wird nun wie bereits beschrieben eingefügt. Wenn wir das Programm nun einmal starten und die Breite der Toolbar verringern, -voilà- präsentiert sie sich wie folgt:



Zum Popup-Menü komme ich gleich, doch zuvor noch ein Hinweis - wieder speziell für die Toolbar: Sie können die Toolbar ja ganz nach Belieben vergrößern oder verkleinern. Dabei kann es passieren, dass einige Buttons nur teilweise zu sehen sind, wenn die Toolbar (eigentlich eher das Band der Rebar) zu klein wird. Da das merkwürdig aussieht, sorgen wir mit dem erweiterten Stilattribut `TBSTYLE_EX_HIDECLIPPEDBUTTONS` dafür, dass solche "abgeschnittenen" Buttons komplett verschwinden:

```
SendMessage(hwndChild,TB_SETEXTENDEDSTYLE,0,
  TBSTYLE_EX_HIDECLIPPEDBUTTONS);
```

3.9.7.1. Praktischer Nutzen

Wie kann man den Chevron nun verwenden? Die Anwendung muss ermitteln, welche der Toolbar-Buttons nicht sichtbar sind und deren Befehle in dem Popup-Menü anbieten. Microsofts PSDK bietet unter dem Indexeintrag "Creating an Internet Explorer-style Toolbar" eine kurze Anleitung ("Handling Chevrons") dafür. Allerdings ist anzumerken, dass sie bei mir nur eingeschränkt funktioniert hat. Ich war sozusagen zu kleinen "Bocksprüngen" gezwungen, bevor ich am Ziel ankam. Doch sehen wir es uns an -

1. Microsoft empfiehlt, zuerst die Maße des Bandes herauszufinden. Das macht Sinn; schließlich müssen wir irgendwie erfahren, welche Buttons der Toolbar sichtbar sind und welche nicht. Es gibt eine Toolbar-Nachricht namens "`TB_ISBUTTONHIDDEN`", die uns hier leider nicht hilft. Unsere Buttons sind zwar nicht zu sehen, allerdings sind sie nicht **versteckt** (im Sinne der Definition der genannten Nachricht). Daher müssen wir anders an die Sache herangehen.
   Microsoft empfiehlt also, mit Hilfe der Nachricht "`RB_GETRECT`" die Abmessungen des Bandes herauszufinden, das den Chevron anzeigt. Das tun wir:

   ```
   SendMessage(hRB,RB_GETRECT,WPARAM(PNMRebarChevron(lp)^.uBand),
     LPARAM(@rc1));
   ```

2. Dann soll man, gemäß der Angabe im PSDK, die Anzahl der Toolbar-Buttons ermitteln. Dafür wird die Nachricht "`TB_BUTTONCOUNT`" verwendet, die sich der Einfachheit halber in einer **for**-Schleife benutzen lässt. Die nächsten Schritte spielen sich also innerhalb dieser Zeilen ab:

   ```
   for i := 0 to SendMessage(hTB,TB_BUTTONCOUNT,0,0) - 1 do begin
     { ... }
   end;
   ```

3. Dann werden die Maße jedes einzelnen Buttons in der Toolbar bestimmt.

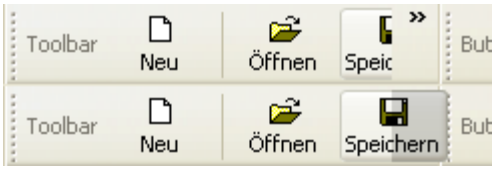
   ```
   SendMessage(hTB,TB_GETITEMRECT,WPARAM(i),LPARAM(@rc2));
   ```

4. Die beiden Rechtecke (das vom Rebar-Control und das der Toolbar) werden nun mit der Funktion "`IntersectRect`" vergleichen. Das Funktionsergebnis interessiert uns hierbei nicht, uns geht es um den ersten Parameter der Funktion: er liefert ein Rechteck (`TRect`) zurück, das sozusagen den gemeinsam genutzten Teil der beiden angegebenen Rechtecke enthält

   ```
   IntersectRect(vis,rc1,rc2);
   ```

   Machen wir ein Beispiel. Im folgenden Bild sehen Sie, dass der "Speichern"-Button nicht vollständig sichtbar ist. Der Übersichtlichkeit wegen habe ich in der unteren Toolbar den Button komplett dargestellt, so dass man auch

sieht, welche Bereiche des Bandes er normalerweise überschreitet:



Die Ermittlung des Button-Rechtecks liefert nun die kompletten Maße zurück, inkl. des nicht sichtbaren Teils (also so wie in der unteren Toolbar). Durch den Vergleich dieses Button-Rechtecks mit dem Rechteck des Rebar-Bandes mit der Funktion "`IntersectRect`", würde aber nur der sichtbare Teil (wie in der oberen Toolbar) zurückgeliefert werden.

5. Dieses zurückgelieferte Rechteck mit dem gemeinsam genutzten Bereich wird nun mit dem originalen Button-Rechteck verglichen, wozu die Funktion "`EqualRect`" benutzt wird. Hierbei interessiert uns der Rückgabewert allerdings wieder, denn wenn die beiden Rechtecke unterschiedlich sind (Ergebnis **false**), bedeutet das, der Button ist nicht oder nicht vollständig zu sehen und muss als Item in das Popup-Menü eingetragen werden. Sind beide Rechtecke identisch (Ergebnis **true**), heißt das selbstverständlich, der Button ist vollständig sichtbar:

```
if(not EqualRect(vis,rc2)) and (tb.fsStyle <> BTNS_SEP) then
  AppendMenu(hm,MF_STRING,tb.idCommand,pchar(pText));
```

Sie sehen hier auch gleich noch die Kontrollabfrage, ob es sich bei dem Button um einen Separator handelt. Der Grund liegt auf der Hand: Separatoren müssen nun wirklich nicht ins Menü. Theoretisch wäre das aber auch kein Problem, da es bei Menüs ja vergleichbares gibt. :o)

Soweit die Anleitung von Microsoft im PSDK. Wenn Sie sie nachvollziehen, dann werden Sie (vielleicht) bemerken, dass die Anzeige der Menüeinträge nicht wirklich funktioniert. Das Beispielprogramm enthält dazu den Compilerschalter `PSDK`, der alle meine zusätzlichen Schritte ausklammert, so dass Sie es gern probieren können.

Folgendes ist zu sagen: die Koordinaten der Toolbar-Buttons werden sich in diesem Experiment nie ändern. Das liegt ganz einfach daran, dass sie relativ zur Toolbar zurückgeliefert werden. Egal wohin Sie die Anwendung auch schieben, der erste Button wird immer die Maße `(0,0,54,36)` besitzen. (Die Werte hängen natürlich von den benutzten Stilattributen ab und können bei Ihnen anders sein.)

Da die Toolbar nun standardmäßig das dritte erzeugte Control ist und sich daher auch im dritten Band (und in der zweiten Zeile) befindet, werden die Abmessungen der Toolbar-Buttons in den meisten Fällen scheinbar außerhalb des Bandes liegen. Die Funktion "`IntersectRect`" würde in solchen Fällen als gemeinsam genutztes Rechteck immer `(0,0,0,0)` liefern.
Für einigermaßen brauchbare Ergebnisse müssten Sie die Toolbar nach ganz oben (an den Anfang der Rebar) schieben, so dass sie zum ersten Control wird. Dann kann es aber passieren (und bei mir ist es passiert!), dass ein Button nicht mehr sichtbar war und trotzdem nicht im Popup-Menü auftauchte. Das lag daran, dass bei der Ermittlung des Band-Rechtecks auch der Platz für den Text und die Breite des Chevrons berücksichtigt wurden. Das heißt also:

1. Nachdem wir im ersten Schritt die Abmessungen des Bandes ermittelt haben, sollten wir herausfinden, welche tatsächlich nutzbare Fläche uns davon zur Verfügung steht. Bedenken Sie, dass die Rebar des Beispielprogramms ja auch noch zusätzlichen Text auf der linken Seite anzeigt. Dieser Bereich ist für Controls nicht nutzbar, wird aber durch Schritt 1 der obigen Anleitung berücksichtigt. Für die Ermittlung der tatsächlich nutzbaren Fläche verwenden wir die Nachricht "`RB_GETBANDBORDERS`"

```
SendMessage(hRB,RB_GETBANDBORDERS,WPARAM(PNMRebarChevron(lp)^.uBand),
  LPARAM(@vis));
```

Leider können wir diese Nachricht nicht allein verwenden, da sie wirklich nur die Grenzwerte liefert. Diese Werte kombinieren wir aber mit dem in Schritt 1 ermittelten Band-Rechteck. Dabei werden die Werte der linken oberen Ecke addiert, die der rechten unteren allerdings subtrahiert, wodurch wir das Band-Rechteck verkleinern:

```
inc(rc1.Left,vis.Left);
inc(rc1.Top,vis.Top);
dec(rc1.Right,vis.Right);
dec(rc1.Bottom,vis.Bottom);
```

Und weil der Chevron auch Platz belegt, der fälschlich als verfügbar betrachtet wird, ziehen wir seine Breite ebenfalls ab:

```
dec(rc1.Right,
  PNMRebarChevron(lp)^.rc.Right - PNMRebarChevron(lp)^.rc.Left);
```

2. Im dritten Schritt der o.g. Anleitung haben wir dann die Maße der einzelnen Toolbar-Buttons ermittelt. Wie ich aber sagte, sind diese relativ zur Toolbar und können sich daher scheinbar außerhalb des Rebar-Bandes befindet - insbesondere im Fall des Standards, wenn sich die Toolbar in der zweiten Reihe des Rebar-Controls befindet. Also müssen wir als einzigen noch erforderlichen Schritt die linke obere Ecke unseres verfügbaren Band-Rechtecks berücksichtigen und auf das Button-Rechteck übertragen:

```
OffsetRect(rc2,rc1.Left,rc1.Top);
```

Glauben Sie es ruhig: nach diesen beiden Ergänzungen funktioniert das Popup-Menü des Chevrons wie gewünscht und zeigt die Toolbar-Buttons an, die momentan nicht zu sehen sind.

Und damit sind wir beim Thema: der Anzeige des Popup-Menüs. Nachdem ich ein bisschen abgeschweift bin, nun zum eigentlichen Kern. Wird der Chevron angeklickt, löst er die Benachrichtigung "`RBN_CHEVRONPUSHED`" aus, die als Teil von "`WM_NOTIFY`" bearbeitet werden kann. Der `lparam` zeigt dabei auf ein Record vom Typ `TNMRebarChevron`. Sie haben in den obigen Zeilen bereits Bekanntschaft damit gemacht. Wir benötigen hier aber noch die Koordinaten des Chevron, die ebenfalls in dem Record mitgeliefert werden. Zur Anzeige des Menüs interessiert uns aber nur die untere linke Ecke.
Die Koordinaten sind allerdings Client-abhängig und müssen erst in gültige Bildschirmwerte umgewandelt werden. Also weisen wir sie der Einfachheit halber einer `TPoint`-Variablen zu:

```
p.X := PNMRebarChevron(lp)^.rc.Left;
p.Y := PNMRebarChevron(lp)^.rc.Bottom;
```

Umgewandelt werden Sie dann mit der Funktion "`ClientToScreen`", von der wir u.a. auch die Anzeige des Menüs abhängig machen:

```
if(ClientToScreen(wnd,p)) and (GetMenuItemCount(hm) > 0) then
  TrackPopupMenu(hm,TPM_LEFTALIGN,p.X,p.Y,0,wnd,nil);
```

TNMRebarChevron-Definition

```
typedef struct tagNMREBARCHEVRON {
    NMHDR hdr;
    UINT uBand;             // Band-Index
    UINT wID;               // Band-ID
    LPARAM lParam;
    RECT rc;                // Chevron-Rechteck
    LPARAM lParamNM;
} NMREBARCHEVRON
```

3.9.8. Einstellungen speichern ... Eine Idee

Als kleines Extra möchte ich noch demonstrieren, wie man das Layout der Rebar-Bänder speichern könnte, so dass sich das Control bei jedem Programmstart immer mit dem gleichen Aussehen präsentiert. Die Betonung liegt dabei auf "könnte", denn diese Vorgehensweise ist keineswegs ein offizieller Weg. Es ist eine Möglichkeit von vielen.

Beginnen wir mit den Grundeinstellungen, die wir zum Laden und Speichern brauchen. Da wäre zuerst ein Record, das unsere Daten aufnehmen soll:

```
type
  rbBandArray = packed record
    Index,
    ID,
    Width : integer;
  end;
```

Dieses Record wird den Index, die ID und die Breite von jeweils einem Rebar-Band aufnehmen. Im Programm nutzen wir dazu ein dynamisches Array, das aus mehreren dieser Records besteht. Außerdem legen wir noch den Registryschlüssel und den Namen des Eintrags fest, unter dem wir diese Werte speichern wollen:

```
const
  szRegKey    = 'Software\Win32-API-Tutorials\Rebar-Demo';
  szValName   = 'RebarBandLayout';
```

3.9.8.1. Das Bandlayout speichern

Beginnen wir mit dem logischen ersten Schritt: dem Speichern der Daten. Idealerweise ist der folgende Code beim Beenden des Programms aufzurufen, damit die Werte auch wirklich erst dann gespeichert werden, wenn der Anwender das Programm schließt.
Zuerst ermitteln wir die Anzahl der Bänder:

```
bIdx := SendMessage(rb,RB_GETBANDCOUNT,0,0);
```

Mit Hilfe einer **for**-Schleife können wir nun die Werte (ID, Stil und Größe) jedes Bandes ermitteln:

```
for i := 0 to bIdx - 1 do begin
  ZeroMemory(@bi,sizeof(bi));
  bi.cbSize     := sizeof(TRebarBandInfo);
  bi.fMask      := RBBIM_ID or RBBIM_SIZE or RBBIM_STYLE;
  SendMessage(rb,RB_GETBANDINFO,i,LPARAM(@bi));
```

Die ermittelten Werte speichern wir in unser dynamisches Array; als Index (sowohl für das Rebar-Band als auch für das jeweilig benutzte Record) dient dabei der aktuelle Schleifenwert:

```
  fba[i].Index := i;
  fba[i].ID    := bi.wID;
  fba[i].Width := bi.cx;
```

Eine kleine Besonderheit ist das Speichern des Stils: wenn ein Rebar-Band in einer neuen Zeile beginnt, dann besitzt es das Attribut `RBBS_BREAK`. Um Speicher zu sparen (und so das Record, das in der Registry gespeichert wird, klein zu halten) habe ich es so gemacht, dass der Index des Bandes als negative Zahl angegeben wird, wenn das Stilattribut gesetzt wird.

```
  if(bi.fStyle and RBBS_BREAK <> 0) then
    fba[i].Index := 0 - fba[i].Index;
end;
```

So kann man Bänder in neuen Zeilen recht einfach unterscheiden, ohne eine weitere Variable im Record nutzen zu müssen.

Das komplette dynamische Array mit dem Bandlayout kann nun in der Registry gespeichert werden. Auf die dazu notwendigen Befehle und ihre Bedeutung möchte ich an der Stelle nicht eingehen, stattdessen würde ich Sie bei evtl. Fragen auf das Registry-Tutorial verweisen. Hier also der Code zum Speichern des Arrays in Kurzform:

```
if(RegCreateKeyEx(HKEY_CURRENT_USER,szRegKey,0,nil,0,
  KEY_READ or KEY_WRITE,nil,reg,nil) = ERROR_SUCCESS) then
try
  RegSetValueEx(reg,szValName,0,REG_BINARY,
    @fba[0],length(fba) * sizeof(rbBandArray));
finally
  RegCloseKey(reg);
end;
```

3.9.8.2. Das Bandlayout laden

Beim Laden gehen wir natürlich den Weg in umgekehrter Reihenfolge: wir lesen zuerst die Daten aus der Registry, bevor wir das Rebar-Control ändern können. Doch auch hier ermitteln wir zuerst die Anzahl der Bänder, weil wir diesen Wert für einen notwendigen Vergleich brauchen werden.
Nach dem Öffnen des Registryschlüssels ermitteln wir aber erst einmal mit Hilfe von "`RegQueryValueEx`", ob der Eintrag überhaupt existiert:

```
if(RegOpenKeyEx(HKEY_CURRENT_USER,szRegKey,0,KEY_READ,
  reg) = ERROR_SUCCESS) then
try
  dwType := REG_NONE;
  dwLen  := 0;

  if(RegQueryValueEx(reg,szValName,nil,@dwType,nil,
    @dwLen) = ERROR_SUCCESS) and
```

So weit, so gut. Da wir die Daten binär gespeichert haben, müssen die vorhandenen Daten (wenn sie denn vorhanden sind) vom gleichen Typ sein:

```
    (dwType = REG_BINARY) and
```

Und selbstverständlich sollten überhaupt Daten vorhanden sein. Ein nützlicher Nebeneffekt des o.g. Registrybefehls ist, dass er den benötigten Speicher zurückliefert, wenn der übergebene Puffer zu klein oder gar Null ist. Das Ergebnis, in der Variablen `dwLen`, sollte also größer als Null sein:

```
    ((dwLen > 0) and
```

Außerdem muss berücksichtigt werden, dass man die Daten in der Registry manuell ändern kann. Wenn ich böswillig bin, dann lösche ich einfach ein paar der gespeicherten Bytes, was sicher unangenehme Folgen für das Programm hat. Nun bietet die folgende Methode zwar keinen vollkommenen Schutz, aber wenn Sie prüfen, ob sich die Anzahl der Bytes ohne Rest durch die Größe des o.g. Records dividieren lässt, können Sie relativ sicher Manipulationen erkennen:

```
     (dwLen mod sizeof(rbBandArray) = 0) and
```

Überlegen Sie bitte: das Record hat eine Grundgröße von 12 Bytes, weil drei Membervariablen (`Index`, `ID` und `Width`) vom Typ `integer` benutzt wurden. Es wird für jedes Band verwendet, das bedeutet bei unserem Beispiel mit seinen 3 Bändern, dass 36 Bytes in der Registry gespeichert werden.
Der Umkehrschluss: wenn Sie 36 Bytes aus der Registry lesen, dann lässt sich dieser Wert ohne Rest durch 12 teilen und ergibt logischerweise 3. Würden Sie die Daten manipulieren und z.B. sechs Bytes entfernen, dann ließen sich die verbliebenen 30 Bytes genau zweimal durch 12 teilen, ergeben aber einen Rest von 6. Die Bedingung stimmt also nicht mehr, und die Daten in der Registry würden auch nicht berücksichtigt werden.
Ein weiterer Nebeneffekt, den wir uns zunutze machen: wenn die Datengröße nach eben definierter Bedingung stimmt, dann sollte die Division durch die Recordgröße logischerweise auch die Anzahl der Bänder des Rebar-Controls ergeben:

```
     (dwLen div sizeof(rbBandArray) = dword(bIdx))) then
```

Nur wenn diese Bedingungen alle erfüllt sind, erzeugen wir ein dynamisches Array mit der benötigten Größe und laden die Daten

```
    begin
      SetLength(fba,dwLen div sizeof(rbBandArray));
      RegQueryValueEx(reg,szValName,nil,@dwType,@fba[0],@dwLen);
    end;
finally
  RegCloseKey(reg);
end;
```

Nachdem wir das dynamische Array gefüllt haben, können wir nun das Rebar-Control "umbauen". Auch hier benutzen wir am besten eine **for**-Schleife, in der wir aber zur Sicherheit prüfen, ob die gespeicherten Indexwerte innerhalb unseres Rebar-Controls liegen, und ob die Suche nach der ID auch wirklich Ergebnisse bringt. (Nur für den Fall, dass jemand die gespeicherten Werte manipuliert hat, ohne dabei die Anzahl der Bytes zu ändern. :o))

```
for i := 0 to length(fba) - 1 do
  if(abs(fba[i].Index) < SendMessage(rb,RB_GETBANDCOUNT,0,0)) and
    (SendMessage(rb,RB_IDTOINDEX,fba[i].ID,0) <> -1) then
  begin
```

Mit Hilfe der Nachricht "`RB_IDTOINDEX`" können nun herausfinden, an welcher Position sich ein Band mit einer bestimmten ID befindet. Für die nächsten Schritte benötigen wir nämlich den Indexwert des Bandes, der sich aber ändern kann. Die einzige Konstante ist demzufolge die ID, die sich nicht ändert - zumindest nicht in diesem Fall:

```
    bIdx        := SendMessage(rb,RB_IDTOINDEX,fba[i].ID,0);
```

Ähnlich wie beim Speichern der Werte holen wir uns die aktuellen Stil- und Größenwerte des Bandes. Die ID interessiert uns diesmal nicht, da wir sie ja bereits durch unser Record kennen:

```
    ZeroMemory(@bi,sizeof(bi));
    bi.cbSize   := sizeof(TRebarBandInfo);
    bi.fMask    := RBBIM_STYLE or RBBIM_SIZE;
    SendMessage(rb,RB_GETBANDINFO,bIdx,LPARAM(@bi));
```

Wir ändern die Breite des Bandes, indem wir den im Record gespeicherten `Width`-Wert an die `cx`-Membervariable des `TRebarBandInfo`-Records übergeben:

```
    bi.cx       := fba[i].Width;
```

Ist der Indexwert kleiner als Null, dann erinnern wir uns daran, dass wir auf diese Weise gespeichert haben, dass ein Band in einer neuen Zeile beginnt. Dementsprechend setzen wir das `RBBS_BREAK`-Attribut (bzw. wir müssen es entfernen, wenn es sich um einen positiven Indexwert handelt):

```
    if(fba[i].Index < 0) then bi.fStyle := bi.fStyle or RBBS_BREAK
      else bi.fStyle := bi.fStyle and not RBBS_BREAK;
```

Diese "gepatchten" Werte werden nun mit der Nachricht "`RB_SETBANDINFO`" an das Rebar-Band zurückgegeben:

```
    SendMessage(rb,RB_SETBANDINFO,bIdx,LPARAM(@bi));
```

Um nun zu guter Letzt das aktuelle Band an seine neue Position zu verschieben, verwenden wir die Nachricht "`RB_MOVEBAND`", der wir den alten und den neuen Index übergeben. Der alte Index ist natürlich der, den wir anfangs ermittelt haben, und der neue wird ja vom dynamischen Array (sprich: von den Registrydaten) bereitgestellt. Angesichts der Tatsache, dass dieser Wert aber auch negativ sein kann, müssen wir die Funktion "`abs`" benutzen, um ihn ohne Vorzeichen zu übergeben:

```
    SendMessage(rb,RB_MOVEBAND,bIdx,abs(fba[i].Index));
  end;
```

Das war´s.

Das Beispielprogramm enthält diese Funktionalität, aber sie ist standardmäßig durch bedingte Kompilierung deaktiviert. Wenn Sie sie verwenden wollen, dann entfernen Sie bitte den Punkt in der Zeile

```
{.$DEFINE LAYOUTSAVING}
```

dadurch werden zusätzliche Funktionen und Anweisungen freigeschaltet, die das Laden und Speichern des Bandlayout demonstrieren. Wenn Sie das Programm nicht mehr verwenden wollen, dann entfernen Sie bitte manuell die angelegten Registryeinträge im Schlüssel `HKEY_CURRENT_USER`.

3.10. Das SysLink-Control

3.10.1. Das SysLink-Control erzeugen

Das SysLink-Control ist ein neues Common Control, das allerdings erst ab Windows XP zur Verfügung steht. Es erfordert außerdem eine Manifestdatei (beiliegend oder in den Ressourcen der Anwendung), damit die Common Controls 6.0 geladen werden können. Andernfalls sehen Sie das Control nicht.
Ihre Delphi-Version sollte natürlich auch aktuell genug sein. Ist dies nicht der Fall, dann nutzen Sie bitte die beiliegende Unit "`CommCtrl_Fragment.pas`", die alle notwendigen Deklarationen enthält.

Das Control bietet eine recht einfache Möglichkeit, Webseiten, Mailadressen usw. im Programm unterzubringen, wobei die typische Linkform genutzt wird. Technisch können Sie es am besten mit einem Label vergleichen, da auch zusätzlicher Text möglich ist:

Bevor Sie das Control erzeugen, müssen Sie die neue Klasse `ICC_LINK_CLASS` mit dem Befehl "`InitCommonControlsEx`" initialisieren:

```
var
  icc: TInitCommonControlsEx = (
    dwSize:sizeof(TInitCommonControlsEx);
    dwICC:ICC_LINK_CLASS;);
begin
  InitCommonControlsEx(icc);

  { ... }
end;
```

Das PSDK sagt sogar, dass der Befehl "`CoInitialize`" aufzurufen ist. Es funktioniert interessanterweise aber auch ohne. Das Erzeugen des Controls geht dann wie üblich mit "`CreateWindowEx`" vonstatten. Als Klassenname ist "`SysLink`" (bzw. die Konstante `WC_LINK`) anzugeben:

```
const
  LINK_TEXT2    : widestring =
    'Hier geht´s zu <a href=" http://www.microsoft.com">Microsoft</a>.' +
    #13#10 + 'Oder?';

hLink2   := CreateWindowExW(0,WC_LINK,pwidechar(LINK_TEXT2),WS_VISIBLE or
WS_CHILD,
  10,60,160,45,wnd,IDC_LCONTROL2,hInstance,nil);
```

An diesem Beispiel sehen Sie, dass die gewünschte URL wie in einer HTML-Seite angegeben wird. Das Control interpretiert in diesem Fall das Tag `href` und stellt den Text dazwischen als Link dar. Zu beachten ist aber, dass ich die Konstante als `widestring` deklariert habe und daher auch "`CreateWindowExW`" benutzen musste.

Neben `href` wird momentan vom Control nur noch `id` unterstützt, das Sie beispielsweise für interne Zwecke innerhalb der Anwendung verwenden können.

Hinweis

Beachten Sie in der o.g. Konstante `LINK_TEXT2` bitte das Leerzeichen vor der URL. Geben Sie es nicht an, wird beim Auslesen der URL komischerweise das H entfernt, so dass aus der Webadresse "ttp://www.microsoft.com" wird. Der Grund für dieses merkwürdige Verhalten ist mir leider unbekannt. Darauf hinweisen möchte ich Sie, auch wenn es sich dabei um einen Denkfehler von mir handeln sollte. (Oder vielleicht gerade deshalb.)

Bei der Benutzung des `id`-Tags passiert das übrigens nicht. Hier ist das Leerzeichen nicht erforderlich, und die ID wird dennoch komplett erkannt.

```
    '<a id="Hallo">Hallo</a>'
```

3.10.2. Mehrere Links in einem Control

Jedes SysLink-Control unterstützt mehr als einen Link. Im obigen Quellcodeausschnitt haben wir lediglich eine Webseite angegeben. Das Beispielprogramm erzeugt aber vorher bereits ein anderes SysLink-Control, das zwei verschiedene `id`-Links enthält.

```
LINK_TEXT1 = 'Oft zitiert: <a id="Hallo">Hallo</a>, ' + #13#10 +
  '<a id="Welt">Welt</a>.';
```

Jeder Link besitzt seinen eigenen internen Indexwert, der in jedem Control bei Null beginnt. Aus diesem Grund besitzen die anfangs gezeigte URL (www.microsoft.com) und der hier zu sehende `id`-Link "Hallo" beide den Index Null.

3.10.3. Klick und Hopp!

Wenn Sie das Control benutzen, wird die Nachricht "`WM_NOTIFY`" an Ihre Anwendung gesendet. Die `code`-Membervariable gibt dabei an, ob es sich um einen Klick oder um das Ergebnis der Enter-Taste handelt:

```
WM_NOTIFY:
  case PNMHdr(lp)^.code of
    NM_RETURN,
    NM_CLICK:
      { ... }
  end;
```

Sie haben nun die Möglichkeit, mit den internen Indexwerten oder mit den IDs und URLs zu arbeiten. Empfehlenswert ist natürlich die Auswertung der Indexes, da sich diese nicht ändern. Im Beispielprogramm wird Ihnen beim Klick auf die Links lediglich die ID bzw. URL angezeigt, daher wollen wir hier ein detailliertes (aber fiktives) Beispiel machen.

Nehmen wir also an, wenn der Anwender auf die beiden Links "Hallo" und "Welt" klickt, sollen interne Programmfunktionen ausgelöst werden, die in dem Fall durch Dialogboxen simuliert werden:

```
if(hwndFrom = hLink1) then
  case PNMLink(lp)^.item.iLink of
    0: MessageBox(wnd,'Aktion 1','Hallo',0);
    1: MessageBox(wnd,'Aktion 2','Welt!',0);
  end;
```

Etwas interessanter wird es allerdings, wenn Sie die IDs direkt vergleichen. Da Sie mit Hilfe des Beispielprogramms die ID des Links "Hallo" ändern können, reagiert dieser also je nach Situation unterschiedlich:

```
if(lstrcmpW(PNMLink(lp)^.item.szId,'Hallo') = 0) then
  MessageBox(wnd,'Aktion 1','Hallo',0)
else if(lstrcmpW(PNMLink(lp)^.item.szId,'Welt') = 0) then
  MessageBox(wnd,'Aktion 2','Welt!',0)
else if(lstrcmpW(PNMLink(lp)^.item.szId,'Morgenstund') = 0) then
  MessageBox(wnd,'Aktion 3','Hallo',0)
else
  MessageBox(wnd,'Link ins Nichts?',nil,MB_ICONWARNING);
```

Die URL aus dem zweiten SysLink-Control leiten Sie dagegen ohne viel Aufhebens an den Standardbrowser Ihres Systems weiter. Da es sich bei der `szUrl`-Variablen um einen Unicode-String handelt, müssen Sie aber die entsprechende Funktion "`ShellExecuteW`" verwenden:

```
if(hwndFrom = hLink2) then
  ShellExecuteW(wnd,nil,PNMLink(lp)^.item.szUrl,nil,nil,SW_SHOWNORMAL);
```

3.10.4. ID und URL eines Links ändern

Zum Ändern einer ID oder URL benötigen wir zuerst eine Variable vom Typ `TLItem`. Dieses Record erwartet zuerst in der `mask`-Membervariablen die Flags, die angeben, was wir ändern wollen. In jedem Fall muss das Flag `LIF_ITEMINDEX` gesetzt werden, da wir den jeweiligen Link über dessen internen Index ansprechen. Wollen wir die ID ändern, geben wir zusätzlich das Flag `LIF_ITEMID` an. Bei der Änderung der URL wäre stattdessen `LIF_URL` zu benutzen:

```
li.mask   := LIF_ITEMINDEX or LIF_ITEMID;
```

Dann geben wir den Index des Links an, den wir ändern wollen:

```
li.iLink  := 0;
```

Nun kopieren wir entweder die neue ID oder die neue URL in den jeweils dafür vorgesehenen Textpuffer des Records. Weil es sich um `widechar`-Arrays handelt, ist "`lstrcpyW`" zu benutzen. Als Beispiel wollen wir einem Link eine neue ID zuweisen und benutzen dafür `szId`:

```
lstrcpyW(li.szId,'Morgenstund');
```

Bei einer neuen URL sähe die Anweisung so aus (beachten Sie, dass hier `szUrl` gefüllt wird):

```
lstrcpyW(li.szUrl,'http://www.borland.com');
```

Die neuen Einstellungen reichen wir dann mit Hilfe der Nachricht "`LM_SETITEM`" an das jeweilige SysLink-Control weiter.

```
SendMessage(hLink1,LM_SETITEM,0,LPARAM(@li));
```

Hinweis
Beachten Sie bitte, dass es auch möglich ist, einem `href`-Link eine neue ID zuzuweisen. Das macht zwar nicht viel Sinn, funktioniert aber.

```
li.mask   := LIF_ITEMINDEX or LIF_ITEMID;
li.iLink  := 0;
lstrcpyW(li.szId,'blabla');
SendMessage(hLink2,LM_SETITEM,0,LPARAM(@li));
```

In dem Fall hätte nun der URL-Link des zweiten SysLink-Controls die ID "blabla". Angezeigt wird allerdings (auf Grund der Auswertung im Beispielprogramm weiterhin die originale URL. Letztlich entscheiden Sie also, womit Sie arbeiten und was Sie ändern.

TLItem-Definition

```
typedef struct tagLITEM {
    UINT mask;
    int iLink;
    UINT state;
    UINT stateMask;
    WCHAR szID[MAX_LINKID_TEXT];
    WCHAR szUrl[L_MAX_URL_LENGTH];
} LITEM
```

3.10.5. Text des SysLink-Controls ändern

Wenn Sie den Text des Controls komplett ändern wollen, dann benutzen Sie dazu die Unicode-Version des bereits bekannten Befehls "SetWindowText" ("SetWindowTextW"):

```
SetWindowTextW(hLink2,
  'Das hier ist <a href=" http://www.borland.com">Borland</a>-Revier.');
```

3.11. Tabsheets

3.11.1. Vorbereiten der Ressourcen

Ich habe mich für das Beispiel für Dialogressourcen entschieden, da es so etwas einfacher wird. Wir brauchen nämlich für jeden Tabsheet ein Fenster bzw. einen Dialog als Container für die jeweiligen Controls. Und dies lässt sich mit Dialogen einfacher bewerkstelligen und ist im Code dann auch übersichtlicher. Wenn Sie auf Dialoge verzichten und Fenster benutzen wollen, dann lautet der Klassenname, den Sie bei "CreateWindowEx" angeben müssen, SysTabcontrol32.

Zusätzlich weisen wir dann im Code jedem "Tabsheet-Dialog" die gleiche Dialog-Prozedur zu, um alle Klicks usw. an einem Ort behandeln zu können.

In dem Ressourcenskript des Beispiels habe ich drei Dialoge definiert: Einen für das Hauptfenster (ID: 100) und zwei mit den Controls für die Tabsheets (ID: 200 und 300). Wichtig ist nun, dass die Dialoge für die Tabsheets die Eigenschaft "Steuerelement" gesetzt haben, damit sie wie Steuerelemente reagieren - sprich: damit man mittels TAB-Taste von einem Reiter zum nächsten springen kann. Im Code des Ressourcenskriptes ist das die fett markierte Einstellung:

```
200 DIALOG DISCARDABLE  0, 0, 145, 95
STYLE DS_CONTROL | WS_CHILD
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        201,10,15,115,12,ES_AUTOHSCROLL
    EDITTEXT        202,10,35,115,12,ES_AUTOHSCROLL
    PUSHBUTTON      "Button1",203,60,60,65,20
END
```

3.11.2. Die Tabsheets erzeugen

Die einzelnen Tabsheets müssen zur Laufzeit dynamisch im Code erzeugt werden. Weil der Quellcode mit einem Tabsheet-Control und mehreren Registerseiten leicht unübersichtlich werden kann, ist es wichtig, von Anfang an effizienten und leicht zu wartenden Code zu schreiben. Aus diesem Grund erzeuge ich die einzelnen Tabsheets in einer Schleife:

```
for i := 0 to CNT_TABS - 1 do
begin
```

In dieser Schleife wird ein paar der benötigten Membervariablen des TCItem-Records gefüllt. Hier ist der Text des Reiters ausreichend, der in pszText angegeben wird. Damit er auch akzeptiert wird, ist die mask-Variable vorher auf TCIF_TEXT zu setzen:

```
  tcItem.mask := TCIF_TEXT;
  tcItem.pszText := pointer(tabSheetCaptions[i]);
```

Das Record wird dann als lParam-Wert mit der Nachricht "TCM_INSERTITEM" an das Tabsheet-Control übergeben:

```
  SendMessage(hTab, TCM_INSERTITEM, i, Integer(@tcItem));
end;
```

Das TCItem-Record enthält natürlich noch ein paar andere Membervariablen:

```
typedef struct tagTCITEM {
  UINT mask;
#if (_WIN32_IE >= 0x0300)
  DWORD dwState;
  DWORD dwStateMask;
#else
  UINT lpReserved1;
  UINT lpReserved2;
#endif
  LPTSTR pszText;
  int cchTextMax;
  int iImage;
  LPARAM lParam;
}
```

mask, pszText, cchTextMax und iImage sind uns in ähnlicher Form bereits von anderen Records bekannt. Als Beispiele sollen stellvertretend die List-View und der Tree-View genannt werden. In aller Kürze: mask gibt an, welche Membervariablen Gültigkeiten haben. pszText ist logischerweise der Text der Seitenreiter, und cchTextMax ist die Anzahl der Textzeichen (diese Variable wird aber nur beim Auslesen der Beschriftungen benötigt). iImage ist der Index eines Bildes einer Imageliste.
Bleiben noch dwState und dwStateMask. Diese beiden Membervariablen sind von der Version des Internet Explorer abhängig. Wie man an der bedingten Compilierung erkennen kann, ist der IE3 das Minimum, um auf die beiden Variablen zugreifen zu können. Das bedeutet, lediglich unter Windows 95 und NT stehen Ihnen diese beiden Variablen nicht zur Verfügung, sondern wären dort durch reservierte (und damit nicht benutzbare) UINT-Variablen ersetzt.

TCM_INSERTITEM-Definition

```
TCM_INSERTITEM
  wParam = (WPARAM) (int) iItem;
  lParam = const (LPARAM) (LPTCITEM) pitem;
```

3.11.3. Tabsheet-Dialoge laden und anzeigen

Es verbleiben noch zwei Schritte: Für jedes Tabsheet einen Dialog aus dem Ressourcenskript laden und den Dialog des ersten Tabsheets anzeigen. Ersteres geschieht wieder in einer Schleife:

```
for i := 0 to length(hTabDlgs) - 1 do
  hTabDlgs[i] := CreateDialog(hInstance, MAKEINTRESOURCE((i + 2) * 100),
  hDlg, @tabdlgfunc);
```

Hier werden also die Dialoge aus der Ressource geladen (clevererweise haben sie die IDs 200 und 300) und **einer** Dialogprozedur zugewiesen. Man kann natürlich auch jedem Dialog eine separate Dialogprozedur zuweisen, aber das ist Geschmackssache.

Zu guter Letzt wollen wir noch den Dialog für den ersten Tabsheet im selbigen anzeigen. Dazu müssen wir dessen Position und Größe ermitteln, was wir mit Hilfe der Nachricht "TCM_GETITEMRECT" tun. Der wParam gibt den Index des Tabsheets an, dessen Position wir benötigen und dem lParam wird ein Zeiger auf ein TRect-Record übergeben, die dann die Position enthält.

```
SendMessage(hTab, TCM_GETITEMRECT, 0, Longint(@rect));
```

Der Funktionsaufruf von "SetWindowPos" sollte keine Probleme bereiten. Er sorgt dafür, dass die jeweilige Registerseite an der richtigen Position erscheint:

```
SetWindowPos(hTabDlgs[0], 0, 50, (rect.Bottom - rect.Top) + 50, 0, 0,
  SWP_NOSIZE or SWP_NOZORDER or SWP_SHOWWINDOW);
```

Eine kleine Anmerkung meinerseits: +50, weil sich meine Dialogelemente nicht am oberen und linken Rand des Dialogfensters befinden.

3.11.4. Wechseln des Seitenreiters

Zuerst einmal das Prinzip: Wie schon gesagt hat jeder Tabsheet ein Dialogfenster als Container für die Controls. Das erspart uns die Arbeit, bei einem Wechsel des Seitenreiters alle Elemente, die sich auf einem Tabsheet befinden, zu verwalten. Wird also nun der Tabsheet gewechselt, gehen wir alle Dialogfenster durch und gucken, was mit ihnen geschen muss: anzeigen oder verbergen. Damit man sieht, wovon ich rede, hier der entsprechende Ausschnitt aus dem Code:

```
// which tabsheet is selected
index := SendMessage(hTab, TCM_GETCURSEL, 0, 0);
```

Mit der Nachricht "TCM_GETCURSEL" ermitteln wir, welcher Tabsheet angeklickt wurde. Die Nachricht hat keine Parameter und liefert uns den Index des Tabsheets zurück.
Dann gehen wir in einer Schleife alle Dialogfenster durch und machen alle unsichtbar, die zu dem nicht ausgewählten Tabsheet gehören - und das Fenster, das zum Tabsheet gehört, wird natürlich sichtbar gemacht:

```
// enum all available tabsheets
for i := 0 to length(hTabDlgs) - 1 do
begin
  // number of tabsheet does not equal selected tabsheet -> hide it
  if i <> index then
    ShowWindow(hTabDlgs[i], SW_HIDE)
  else // else make dialog visible
  begin
    SendMessage(hdr^.hwndFrom, TCM_GETITEMRECT, i,
      Longint(@rect));
    SetWindowPos(hTabDlgs[index], 0, 50, (rect.Bottom - rect.Top)
      + 50, 0, 0, SWP_NOSIZE or SWP_NOZORDER or SWP_SHOWWINDOW);
  end;
end;
```

Wie bekommen wir nun mit, dass ein Tabsheet angeklickt wurde? Das ist recht einfach: klickt der Benutzer einen Seitenreiter an, bekommen wir eine "WM_NOTIFY"-Nachricht. In der NMHDR-Struktur erhalten wir als Benachrichtigungscode "TCN_SELCHANGE" für einen Klick auf einen Seitenreiter:

```
WM_NOTIFY:
  begin
    hdr := PNMHDR(lParam);
    case hdr^.code of
      TCN_SELCHANGE:
        // tabsheet selection has changed
        { ... }
    end;
  end;
```

4. Systemfunktionen

4.1. Der Timer ohne die VCL

4.1.1. Den Timer erzeugen

Bei der VCL ist es ganz leicht: Sie ziehen die **Timer**-Komponente auf Ihr Formular, stellen das Intervall ein, klicken doppelt drauf ... und schreiben den Code, der nach Erreichen des Intervalls ausgeführt werden soll. Viel schwerer ist es mit dem API aber auch nicht.

Mit der Funktion "`SetTimer`" stellt uns das API einen Timer zur Verfügung. Wir brauchen eigentlich nur das Handle des übergeordneten Fensters, eine ID für den Timer und das Zeitintervall (in Millisekunden). Der vierte Parameter der Funktion ist für uns in dem Fall nicht von Interesse. Rückgabewert ist dann das Handle des Timers:

```
hTimer := SetTimer(hWnd, IDC_TIMER, {Intervall von 1sec ->} 1000, nil);
```

4.1.2. Auf das Zeitintervall reagieren

Was in der VCL-Programmierung das "**OnTimer**"-Event ist, ist beim API die Nachricht "`WM_TIMER`", die gesendet wird, sobald das angegebene Intervall abgelaufen ist. Gemäß der Deklaration aus unserem Beispiel würde die Nachricht also bereits nach 1 Sekunde das erste Mal auftreten ... und so fort ... Wir müssen also nichts weiter tun, als diese Nachricht in der Nachrichtenfunktion abzufangen und zu bearbeiten:

```
WM_TIMER:
  begin
    {Countdown aktualisieren}
    Dec(i);
    wvsprintf(buffer, 'Countdown %d', PChar(@i));
    SetWindowText(hwndLabel, buffer);
    ...
  end;
```

Da wir im Beispielprogramm nur einen Timer benutzen, entfällt die gezielte Abfrage der Timer-ID. Hätten Sie z.B. mehrere Timer in Ihrem Programm, dann müssten Sie den Wert von `wParam` abfragen, der die ID enthält:

```
WM_TIMER:
  case wParam of
    IDC_TIMER1:
      // Timer 1 ist aktiv geworden
    IDC_TIMER2:
      // Timer 2 ist aktiv geworden

    // usw.
  end;
```

4.1.3. Den Timer freigeben

Wenn Sie den Timer nicht mehr benötigen, bzw. generell beim Beenden des Programms, müssen Sie ihn freigeben ("zerstören"). Dazu dient die Funktion "`KillTimer`", die das Handle des übergeordneten Fensters und die Timer-ID als Parameter erwartet:

```
KillTimer(hWnd, IDC_TIMER);
```

4.2. Verbindung zur Taskbar Notification Area

4.2.1. Grundlagen

In diesem Tutorial wollen wir Verbindung zur TNA ("Taskbar Notification Area") aufnehmen. Sie kennen sicher einige Programme, die unsichtbar im Hintergrund laufen und nur durch das kleine Icon im Bereich der Uhr sicht- und steuerbar sind. Des Weiteren zeigt dieses Tutorial, wie Sie die neuen Balloon-Tipps verwenden können.

Die Grundlage für unser Beispiel ist ein unsichtbares Fenster, das wir im Hauptteil registrieren und erzeugen. Wir benötigen es nur, um eine Nachrichtenverarbeitung zu haben, denn wir müssen ja z.B. auf das TNA-Symbol zugreifen und Nachrichten von dort verarbeiten können. Besondere Stilattribute oder Größenangaben sind dafür nicht erforderlich.

Beispiel

```
var
  wc: TWndClassEx = (
    cbSize          : sizeof(TWndClassEx);
    Style           : 0;
    lpfnWndProc     : @WndProc;
    cbClsExtra      : 0;
    cbWndExtra      : 0;
    hIcon           : 0;
    hCursor         : 0;
    hbrBackground   : 0;
    lpszMenuName    : nil;
    lpszClassName   : szClassname;
    hIconSm         : 0;
  );
  msg: TMsg;

begin
  wc.hInstance  := hInstance;
  RegisterClassEx(wc);
  CreateWindowEx (0, szClassname, szClassname, 0, 0, 0, 0, 0, 0, 0, hInstance,
nil);

  // Nachrichtenschleife
end;
```

4.2.2. Das TNA-Icon erzeugen

Um ein Symbol in der TNA ablegen zu können, brauchen wir zuerst das `TNotifyIconData`-Record. Nachdem wir die Unit "`ShellAPI.pas`" eingebunden haben, können wir die meisten Variablen bereits während der Initialisierung füllen:

```
var
  NID : TNotifyIconData = (
    cbSize           : sizeof(TNotifyIconData);
    uID              : 1052002;
    uFlags           : NIF_MESSAGE or NIF_ICON or NIF_TIP;
    uCallbackMessage : WM_TNAMSG;
    hIcon            : 0;
    szTip            : szClassname;
  );
```

Ich habe im Beispiel als Wert für `uID` das Datum gewählt. Sie können nach diesem Prinzip vorgehen und jedem Ihrer TNA-Programme eine eindeutige ID zuordnen. Sie können ebenso aber auch den Wert Null benutzen. Wichtig ist, dass Sie für jedes TNA-Symbol, das Ihr Programm verwendet, eine eigene ID benutzen, da dies sozusagen der Kanal ist, auf dem das Betriebssystem und Ihr Programm die einzelnen Symbole ansprechen. Die Variable `uCallbackMessage` besitzt eine Nachricht namens "`WM_TNAMSG`" als Wert. Diese Nachricht existiert nicht im System, sie vom Programm definiert:

```
const
  WM_TNAMSG  = WM_USER + 10;
```

Außerdem sei noch ein Wort zu den Flags gestattet:

Wert	Bedeutung
NIF_MESSAGE	wir wollen Nachrichten bearbeiten, also muss `uCallbackMessage` eine gültige Variable bezeichnen
NIF_ICON	wir wollen ein Icon sehen, also muss ein `hIcon` gültiges Symbol-Handle sein
NIF_TIP	ein Tooltipp soll zu sehen sein

TNotifyIconData-Definition

```
typedef struct _NOTIFYICONDATA {
    DWORD cbSize;             // Größe des Record
    HWND hWnd;                // Handle des Parent-Fensters
    UINT uID;                 // eindeutige ID
    UINT uFlags;              // Flags
    UINT uCallbackMessage;    // Nachrichtenbezeichner
    HICON hIcon;              // Icon
    char szTip[64];           // Tooltipp
} NOTIFYICONDATA
```

Es fehlen noch zwei Dinge, damit unser Kontakt zustande kommt: wir müssen das Fensterhandle und ein Symbol festlegen. Da wir das Handle zum Zeitpunkt der Variablen-Initialisierung nicht kennen (können), weisen wir es während der Erzeugung zu:

```
WM_CREATE:
  begin
    NID.wnd   := wnd; // Fenster-Handle
    NID.hIcon := LoadIcon(0,IDI_INFORMATION); // Icon laden

    Shell_NotifyIcon(NIM_ADD,@NID); // Icon anzeigen
  end;
```

In der letzten Zeile sehen Sie auch gleich den Befehl, der den Kontakt unseres Programmes zur "Taskbar Notification Area" herstellen, aber auch lösen kann. Folgende Aktionen sind möglich:

Wert	Bedeutung
NIM_ADD	das Symbol wird hinzugefügt
NIM_DELETE	das Symbol wird entfernt
NIM_MODIFY	das Symbol wird geändert (das gilt auch für den Tooltipp)

Bevor Sie das Programm also testen, sollten Sie dafür sorgen, dass das Symbol beim Beenden wieder entfernt wird. Wenn Sie das vergessen, dann entfernt das System das Symbol, sobald Sie mit dem Mauszeiger drauf kommen. Aber es sieht unschön aus, also sollten Sie daran denken:

```
WM_DESTROY:
  begin
    Shell_NotifyIcon(NIM_DELETE,@NID); // Icon entfernen
    PostQuitMessage(0);
  end;
```

Shell_NotifyIcon-Definition

```
WINSHELLAPI BOOL WINAPI Shell_NotifyIcon(
    DWORD dwMessage,         // Message value to send
    PNOTIFYICONDATA pnid     // address of a NOTIFYICONDATA structure
);
```

4.2.3. Interaktion mit dem Anwender

Um auf die Aktionen des Anwenders zu reagieren, benötigen wir jetzt unsere eigene Nachricht "WM_TNAMSG":

```
WM_TNAMSG:
  case lp of
    WM_RBUTTONUP:
      begin
        // Popupmenü erzeugen
      end;
  end;
```

Klickt der Anwender also mit der rechten Maustaste auf unser Symbol, dann wird das Popupmenü erzeugt und angezeigt, und er hat die Auswahl. Beachten Sie, dass wir hier "WM_RBUTTONUP" benutzen. Das heißt, der Anwender muss die rechte Maustaste erst wieder loslassen; erst dann sieht er das Menü.

Zu guter Letzt wollen wir sowohl Symbol als auch Tooltipp-Text ändern. Dafür laden wir einfach nur ein neues Icon, definieren einen neuen Text und modifizieren das Symbol. Und all das passiert bei einem Doppelklick mit der linken Maustaste:

```
WM_TNAMSG:
  case lp of
    WM_LBUTTONDBLCLK:
      begin
        NID.hIcon := LoadIcon(0,IDI_WARNING);
        lstrcpy(NID.szTip,'geänderter Tooltipp-Text');

        Shell_NotifyIcon(NIM_MODIFY,@NID);
      end;
  end;
```

4.2.4. Die Anwendung in die TNA minimieren

In diesem Kapitel soll es hauptsächlich darum gehen, wie man die eigene Anwendung in die "Taskbar Notification Area" minimieren lassen kann. In Foren wird ab und zu gefragt, wie man das bewerkstelligen kann. Bei nonVCL-Programmen ist dies relativ einfach. Beim Erzeugen des Fensters laden wir das gewünschte Symbol und weisen das Fensterhandle zu (ganz so, wie es in den letzten Kapiteln demonstriert wurde):

```
  WM_CREATE:
    begin
      NID.wnd   := wnd;
      NID.hIcon := LoadIcon(hInstance,'JIM');
    end;
```

Und beim Beenden des Programms entfernen wir sicherheitshalber das TNA-Icon (auch so, wie es bereits besprochen wurde):

```
  WM_DESTROY:
    begin
      Shell_NotifyIcon(NIM_DELETE,@NID);

      { ... }
    end;
```

So weit nichts Neues.

Um auf das Minimieren des Fensters zu reagieren, gibt es zwei Wege, die wir gehen können. Einer wäre, die Nachricht "WM_SYSCOMMAND" zu bearbeiten, wenn der wParam den Wert "SC_MINIMIZE" enthält. Allerdings reagiert unser Programm dann nicht auf den globalen System-Hotkey WIN+M, mit dem man recht bequem alle Fenster minimieren kann (sofern Ihre Tastatur die Taste mit dem Windows-Logo enthält, natürlich!). Das heißt, unser Programm reagiert schon - aber es wird wirklich nur minimiert, nicht aber in die TNA "verschoben".

Besser ist daher die Bearbeitung der Nachricht "WM_SIZE", deren wParam den Wert "SIZE_MINIMIZED" enthält, wenn das Fenster minimiert wurde. **Und** diesmal funktioniert es auch mit den angesprochenen System-Hotkey:

```
WM_SIZE:
  if(wp = SIZE_MINIMIZED) then begin
    if(Shell_NotifyIcon(NIM_ADD,@NID)) then
      ShowWindow(wnd,SW_HIDE);
  end else
    Result := DefWindowProc(wnd,uMsg,wp,lp);
```

Zu beachten ist allerdings, dass die Nachricht "WM_SIZE" auch ausgelöst wird, wenn das Fenster maximiert oder anderweitig in seiner Größe verändert wird. Darum ja auch die Einschränkung im Code, mit der wir gezielt auf die Minimierung prüfen. Dennoch dürfen wir die die anderen Möglichkeiten nicht blockieren und müssen Sie deshalb an die Standardfensterprozedur "DefWindowProc" weiterleiten.

Wie dem auch sei, das Programm (s. Beispiel "OnMinimize.dpr") weist nun das gewünschte Verhalten auf und minimiert sich in die TNA.

Die VCL-Version vs. Delphi 5

Wollen Sie das gleiche Verhalten für Ihre VCL-Programme verwenden, dann liegt es auf der Hand das obige Verhalten entsprechend nachzubilden. Sie müssen also auch hier auf die Nachricht "WM_SIZE" reagieren, was beispielsweise so aussehen kann:

```
type
  TForm1 = class(TForm)
    { ... }
  private
    procedure WMSize(var Message: TWMSize); message WM_SIZE;
  end;
```

In dieser privaten Prozedur erstellen Sie ebenfalls das TNA-Symbol und lassen den Button in der Startleiste verschwinden. Bei VCL-Programmen ist allerdings das TApplication-Objekt für diesen Button verantwortlich. Das Formular wird durch die Minimierung selbst bereits versteckt, so dass Sie hier also das Handle von TApplication benutzen müssen:

```
procedure TForm1.WMSize(var Message: TWMSize);
begin
  if(Message.Msg = WM_SIZE) and
    (Message.SizeType = SIZE_MINIMIZED) then
  begin
    if(Shell_NotifyIcon(NIM_ADD,@NID)) then
      ShowWindow(Application.Handle,SW_HIDE)
    else
      inherited;
  end else
    inherited;
end;
```

Eine weitere Möglichkeit wäre, direkt in die "WndProc" des Formulars einzugreifen und dort die o.g. Nachricht zu bearbeiten, etwa:

```
type
  TForm1 = class(TForm)
    { ... }
  protected
    procedure WndProc(var Message: TMessage); override;
  end;
```

Hierbei müssen Sie allerdings beachten, dass diese Prozedur wie in einem nonVCL-Programm der zentrale Anlaufpunkt für alle Arten von Nachrichten ist. Sie dürfen also keine blockieren, sondern Sie dürfen nur die gewünschte bearbeiten

und **müssen** alle anderen weiterleiten.
Dabei kommt es auf die Art und Weise an, wie Sie das tun. In meinem Beispiel wird die Standardprozedur am Ende generell aufgerufen, so dass bei einer evtl. Nachrichtenbearbeitung die Prozedur vorher mit "`exit`" verlassen werden sollte:

```
procedure TForm1.WndProc(var Message: TMessage);
begin
  case Message.Msg of
    WM_SIZE:
      if(Message.wParam = SIZE_MINIMIZED) then begin
        if(Shell_NotifyIcon(NIM_ADD,@NID)) then begin
          ShowWindow(Application.Handle,SW_HIDE)
          exit;
        end;
      end;

    { ... }
  end;

  inherited WndProc(Message);
end;
```

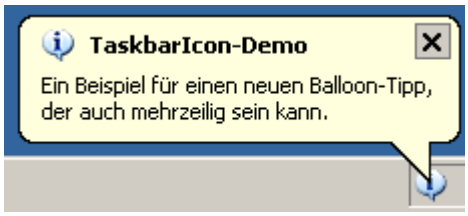
So weit die Theorie. In der Praxis scheiterten beide Versuche bei mir (Delphi 5 Pro) allerdings daran, dass `SIZE_MINIMIZED` ignoriert wird. Warum auch immer ...

Wenn Sie das Flag testweise durch sein Gegenstück `SIZE_MAXIMIZED` (= die Anwendung wurde maximiert) ersetzen, dann erscheint das TNA-Symbol, und der Button in der Startleiste verschwindet. Wie gesagt: theoretisch sollte es also auch beim Minimieren funktionieren ...
Möglicherweise handelt es sich hierbei aber auch "nur" um einen Bug von Delphi 5. Vielleicht ist das in Ihrer Delphi-Version auch anders bzw. (falls es ein Bug ist:) behoben, so dass Sie obige Anregungen ausprobieren Ihre Ergebnisse im Support-Forum vorstellen können.

Jedenfalls wollte ich Ihnen den Vorschlag nicht vorenthalten, auch wenn er bei mir nicht funktioniert.

4.2.5. Balloon-Tipps

Wenn Sie min. die Version 5.0 der Datei "`shell32.dll`" besitzen, können Sie das `TNotifyIconData`-Record erweitern und die neuen Balloon-Tipps verwenden:



Windows ME, 2000 und XP gehören daher zu den Betriebssystemen, mit denen man das folgende Beispiel nachvollziehen kann. Ein Update des Internet Explorer reicht in diesem Fall (laut PSDK) nicht aus, um die neue DLL-Version auch unter Windows 95, 98 und NT einzuspielen. Ich richte mich also nach den Vorgaben von Microsoft und übernehme keine Garantie für Experimente dieser Art.

Hinweis

Diesem Tutorial liegt eine spezielle Unit, "`ShellAPI_Fragment.pas`", bei. Sie enthält die notwendigen Erweiterungen zum Nachvollziehen dieses Beispiels und wurde erfolgreich mit Delphi 5 Professional getestet. Das Beispielprogramm ist daher mit einem Compilerschalter versehen, durch den die Unit nur bei Delphi 5 berücksichtigt wird.
Prüfen Sie bitte vorher, ob die Erweiterungen für Ihre Delphi-Version überhaupt noch erforderlich sind. Probieren Sie

ggf. aus, ob sich das Programm mit Ihrer Delphi-Version kompilieren lässt, wenn Sie die o.g. spezielle Unit verwenden.

4.2.5.1. Die "ShellAPI.pas" patchen

Zuerst ist ein Eingriff in die Unit "`ShellAPI.pas`" erforderlich, da Borland meines Wissens nach auch bei Delphi 6 noch die alte Version ausliefert. Wir benötigen zunächst einen Einblick in das das neue `TNotifyIcon`-Record. Dabei hilft uns eine aktuelle Version des MSDN oder PSDK. Ich demonstriere die Änderungen nachfolgend an der Ansi-Version. Die Unicode-Version (`_NOTIFYICONDATAW`) muss analog geändert werden, allerdings ist dort `AnsiChar` durch `WideChar` zu ersetzen.

```
type
  _NOTIFYICONDATAA = record
    cbSize: DWORD;
    Wnd: HWND;
    uID: UINT;
    uFlags: UINT;
    uCallbackMessage: UINT;
    hIcon: HICON;
    szTip: array [0..127] of AnsiChar;
    dwState: DWORD;
    dwStateMask: DWORD;
    szInfo: array[0..255]of AnsiChar;
    case integer of
      0: (uTimeout: UINT);
      1: (uVersion: UINT;
          szInfoTitle: array[0..63]of AnsiChar;
          dwInfoFlags: DWORD;
          guidItem : TGuid;
         )
  end;
```

Und ein paar neue Konstanten sind ebenfalls erforderlich, damit wir die erweiterten Funktionen nutzen können:

```
const
  {$EXTERNALSYM NIN_SELECT}
  NIN_SELECT     = WM_USER + 0;
  {$EXTERNALSYM NINF_KEY}
  NINF_KEY       = $01;
  {$EXTERNALSYM NIN_KEYSELECT}
  NIN_KEYSELECT  = NIN_SELECT or NINF_KEY;
  {$EXTERNALSYM NIN_BALLOONSHOW}
  NIN_BALLOONSHOW      = WM_USER + 2;
  {$EXTERNALSYM NIN_BALLOONHIDE}
  NIN_BALLOONHIDE      = WM_USER + 3;
  {$EXTERNALSYM NIN_BALLOONTIMEOUT}
  NIN_BALLOONTIMEOUT   = WM_USER + 4;
  {$EXTERNALSYM NIN_BALLOONUSERCLICK}
  NIN_BALLOONUSERCLICK = WM_USER + 5;

  {$EXTERNALSYM NIM_SETFOCUS}
  NIM_SETFOCUS    = $00000003;
  {$EXTERNALSYM NIM_SETVERSION}
  NIM_SETVERSION  = $00000004;
  {$EXTERNALSYM NOTIFYICON_VERSION}
  NOTIFYICON_VERSION = 3;

  {$EXTERNALSYM NIF_STATE}
  NIF_STATE       = $00000008;
  {$EXTERNALSYM NIF_INFO}
  NIF_INFO        = $00000010;
  {$EXTERNALSYM NIF_GUID}
  NIF_GUID        = $00000020;
  {$EXTERNALSYM NIS_HIDDEN}
  NIS_HIDDEN      = $00000001;
  {$EXTERNALSYM NIS_SHAREDICON}
  NIS_SHAREDICON   = $00000002;

// says this is the source of a shared icon

// Notify Icon Infotip flags
  {$EXTERNALSYM NIIF_NONE}
  NIIF_NONE        = $00000000;
// icon flags are mutualy exclusive
// and take only the lowest 2 bits
  {$EXTERNALSYM NIIF_INFO}
  NIIF_INFO        = $00000001;
  {$EXTERNALSYM NIIF_WARNING}
  NIIF_WARNING     = $00000002;
  {$EXTERNALSYM NIIF_ERROR}
  NIIF_ERROR       = $00000003;
  {$EXTERNALSYM NIIF_ICON_MASK}
  NIIF_ICON_MASK   = $0000000F;
  {$EXTERNALSYM NIIF_NOSOUND}
  NIIF_NOSOUND     = $00000010;
```

Hinweis
Durch die Verwendung von "`WM_USER`" muss die Unit "`Messages.pas`" zusätzlich eingebunden werden. Und man sollte für die eigene TNA-Nachricht möglichst nicht mehr den Bereich von "`WM_USER`" bis "`WM_USER + 5`" verwenden, da diese Werte von den neuen Nachrichten benutzt werden.

Und damit wir immer die korrekte Recordgröße verwenden, definieren wir noch die folgenden Konstanten. Diese enthalten die Größe des Originalrecord ohne die o.g. Erweiterungen, und die Größe des Records ohne die `TGuid`-Membervariable:

```
const
  {$EXTERNALSYM NOTIFYICONDATAA_V1_SIZE}
  NOTIFYICONDATAA_V1_SIZE = 88;
  {$EXTERNALSYM NOTIFYICONDATAW_V1_SIZE}
  NOTIFYICONDATAW_V1_SIZE = 152;
  {$EXTERNALSYM NOTIFYICONDATA_V1_SIZE}
  NOTIFYICONDATA_V1_SIZE  = NOTIFYICONDATAA_V1_SIZE;

  {$EXTERNALSYM NOTIFYICONDATAA_V2_SIZE}
  NOTIFYICONDATAA_V2_SIZE = sizeof(NOTIFYICONDATAA) - (sizeof(TGUID));
  {$EXTERNALSYM NOTIFYICONDATAW_V2_SIZE}
  NOTIFYICONDATAW_V2_SIZE = sizeof(NOTIFYICONDATAW) - (sizeof(TGUID));
  {$EXTERNALSYM NOTIFYICONDATA_V2_SIZE}
  NOTIFYICONDATA_V2_SIZE  = NOTIFYICONDATAA_V2_SIZE;
```

4.2.5.2. Das Programm mit Balloon-Tipps erweitern

Zuerst sollten wir herausfinden, welche Version der "`shell32.dll`" vorhanden ist. Dazu nutzen wir eine Funktion, die die Bibliothek selbst exportiert:

```
HRESULT CALLBACK DllGetVersion(
  DLLVERSIONINFO* pdvi // Zeiger auf "DLLVERSIONINFO"-Record
);
```

Das benötigte Record, `DLLVERERSIONINFO` finden Sie in der Unit "`DllVersion.pas`", die vom Beispielprogramm benutzt wird.

DLLVERERSIONINFO-Definition

```
typedef struct _DllVersionInfo {
    DWORD  cbSize;
    DWORD  dwMajorVersion;
    DWORD  dwMinorVersion;
    DWORD  dwBuildNumber;
    DWORD  dwPlatformID;
} DLLVERSIONINFO;
```

In unserem Fall werden die Haupt- und Nebenversionsnummer zu einem numerischen Wert zusammengefasst. So steht 500 beispielsweise für die Version 5.0; 600 würde demzufolge Version 6.0 bedeuten usw.

```
dll := LoadLibrary('shell32.dll');
if(dll <> 0) then begin
  DllGetVersion := GetProcAddress(dll,'DllGetVersion');
  if(@DllGetVersion <> nil) then begin
    ZeroMemory(@ver,sizeof(TDllVersionInfo));
    ver.cbSize := sizeof(TDllVersionInfo);
    if(DllGetVersion(@ver) = NOERROR) then
      shell32_ver := (ver.dwMajorVersion * 100) + ver.dwMinorVersion;
  end;

  FreeLibrary(dll);
end;
```

Bei der Initialisierung setzen wir die Versionsnummer auf 400, für den Fall, dass wir mit einer DLL-Version arbeiten, die die o.g. Funktion noch nicht exportiert, bzw. keinen Wert zurückliefert o.ä. Wir müssen ja leider auch immer mit solchen Dingen rechnen.

Dann setzen wir, abhängig von der vorhandenen DLL-Version die Größe des Records:

```
if(shell32_ver = 600) then NID.cbSize := sizeof(TNotifyIconData)
  else if(shell32_ver >= 500) then NID.cbSize := NOTIFYICONDATA_V2_SIZE
    else NID.cbSize := NOTIFYICONDATA_V1_SIZE;
```

Hier sehen Sie die Anwendung der Konstanten, die wir hier deklariert haben. So können wir z.B. unter Windows XP (DLL-Version 6.0) das komplette Record benutzen. Unter Windows ME und 2000 (DLL-Version 5.0) steht uns immerhin alles außer der `TGuid`-Membervariablen zur Verfügung, weshalb wir die Konstante `NOTIFYICONDATA_V2_SIZE` nutzen müssen. Und unter Windows 95, 98 und NT müssen wir das Originalrecord und seine Größe benutzen, da uns hier die erweiterten Funktionen natürlich nicht zur Verfügung stehen - und das entspricht der Konstanten `NOTIFYICONDATA_V1_SIZE`.

Was passiert dabei?

Im Fall von `NOTIFYICONDATA_V2_SIZE` wird die Größe des erweiterten Records lediglich um die Größe der `TGuid`-Variablen verringert. Ein Aufruf unter Windows 2000 beispielsweise würde demzufolge dann auch nur die noch vorhandenen Membervariablen füllen.

`NOTIFYICONDATA_V1_SIZE` hat dagegen zwei feste Werte (jeweils für die Ansi- und Unicode-Version), die sich aus dem Originalrecord errechnen. Im Fall der Ansi-Version wären das z.B. 88 Bytes. Auf diesen Wert würde die Größe des Records eingestellt werden. Ein Aufruf würde dann also auch nur die ersten 88 Bytes berücksichtigen. Und das bedeutet auch, dass 64 Bytes von der `szTip`-Variablen abgezogen werden. Probleme gibt es daher unter keinem Betriebssystem, selbst wenn Sie unter Windows 98 mit dem neuen Record arbeiten. Sie sollten nur nicht auf Membervariablen zugreifen, die erst durch neuere DLL-Versionen eingeführt werden. Im Idealfall passiert nichts, ebenso wäre aber auch eine Fehlermeldung oder ein Absturz denkbar.

Eine Sicherheitsabfrage oder -prüfung sollte daher Pflicht sein. Nehmen wir als Beispiel den Balloon-Stil, der sich ja erst ab DLL-Version 5.0 oder höher nutzen lässt, und der im Beispielprogramm auch nur dann verwendet wird:

```
if(shell32_ver >= 500) then begin
  if(BalloonsEnabled) then begin
    NID.uFlags      := NID.uFlags or NIF_INFO;
    NID.szInfo      := 'Ein Beispiel für einen neuen Balloon-Tipp,' +
      #13#10 + 'der auch mehrzeilig sein kann.';
    NID.szInfoTitle := szClassname;
    NID.dwInfoFlags := NIIF_INFO;
  end else
    MessageBox(wnd,'Balloon-Tipps sind bei Ihnen deaktiviert!',
    nil,MB_ICONERROR)
end;
```

Wenn wir das Programm nun starten und die DLL aktuell genug ist, sehen wir den neuen Balloon-Tipp. Andernfalls passiert aber auch nichts, und wir haben zumindest den gewohnten TNA-Funktionsumfang.

Hinweis
Das System kümmert sich um die Zeitdauer der Anzeige. Im PSDK heißt es dazu sinngemäß, dass ein Balloon-Tipp mit einem Timeout von 30 Sekunden für ca. 7 Sekunden sichtbar ist, sobald eine zweite Anwendung einen Tipp anzeigen möchte. Der Timeout-Wert, den Sie angeben, dient also mehr als Richtwert und weniger als tatsächliche Anzeigedauer.

4.2.5.3. Interaktion mit dem Anwender II

Zum erweiterten Funktionsumfang gehören auch neue Nachrichtenwerte, die unserer eigenen TNA-Nachricht als Teil des `lparam`-Wertes übergeben werden. So können wir z.B. auf das Erscheinen und Verschwinden des Balloon-Tipp reagieren:

```
case uMsg of
  WM_TNAMSG:
    case lp of
      NIN_BALLOONSHOW:
        MessageBox(0,'I see the balloon! :o)',szClassname,
          MB_OK or MB_ICONINFORMATION);
      NIN_BALLOONHIDE:
        MessageBox(0,'It´s gone ... :o(',szClassname,
          MB_OK or MB_ICONINFORMATION);
    end;
end;
```

Hinweis

Die Nachricht "NIN_BALLOONHIDE" wird z.B. beim Entfernen des TNA-Icons gesendet - sofern der Tipp dabei noch zu sehen ist. Ein Mausklick und der Timeout lösen stattdessen die Nachrichten "NIN_BALLOONUSERCLICK" bzw. "NIN_BALLOONTIMEOUT" aus (wie Sie im Beispielprogramm sehen können).

Achtung!

Dem PSDK und dem MSDN zufolge kann man die Art und Weise der Benachrichtigung mit Hilfe der Anweisung "NIM_SETVERSION" einstellen. Zur Auswahl hat man das Verhalten der Version 5.0 bzw. der Vorgängerversion:

```
NID.uVersion := NOTIFYICON_VERSION; // Use the Windows 2000 behavior
Shell_NotifyIcon(NIM_SETVERSION,@NID);
```

Die Alternative wäre der Wert Null ("Windows 95 behavior"). Ich gehe aber davon aus, dass durch die Nutzung der erweiterten Funktionalität automatisch auch die Benachrichtungsart geändert wird. Lässt man nämlich diese beiden Zeilen weg, kann man trotzdem die o.g. Nachrichten bearbeiten. Nur darauf verlassen sollte man sich nicht unbedingt.

4.3. Hotkeys und Shortcuts

4.3.1. Systemweite Hotkeys

4.3.1.1. Einen Hotkey registrieren

Durch die Arbeit mit Fenstern und Dialogen kennen Sie bereits die Shortcuts, die Sie dort definieren können. Wenn z.B. ein Buchstabe auf einem Button, in einer Checkbox usw. unterstrichen ist, dann erwartet der Anwender, dass die entsprechende Aktion durch Drücken der ALT-Taste in Verbindung mit diesem Buchstaben ausgelöst wird.
Es ist aber auch möglich, systemweite Hotkeys zu definieren, die relativ unabhängig von ihrem Programm sind. Das heißt, notwendig ist Ihr Programm schon (der Hotkey muss ja erzeugt werden), aber es muss nicht unbedingt aktiv sein. Es kann im Hintergrund bleiben und z.B. durch den Hotkey nach vorn geholt werden.

In diesem Beispiel wollen wir einen Hotkey definieren und benutzen. Grundlage dafür ist die API-Funktion "`RegisterHotKey`". Neben dem Handle des Fensters als ersten Parameter, geben Sie als zweites eine möglichst eindeutige ID an, mit der der Hotkey später identifiziert werden kann. Der dritte Parameter kann eine oder mehrere der folgenden Werte enthalten, die mit **xor** verknüpft werden:

Wert	Bedeutung
MOD_ALT	die ALT-Taste muss gedrückt gehalten sein
MOD_CONTROL	die STRG-Taste muss gedrückt gehalten sein
MOD_SHIFT	die Shift-Taste muss gedrückt gehalten sein
MOD_WIN	die Windows-Taste muss gedrückt gehalten sein

Zuletzt geben wir die eigentliche Taste an, die unserem Hotkey zugeordnet werden soll. Wollen wir z.B. den Hotkey STRG+ALT+H registrieren, dann würde der Befehl so aussehen:

```
if(RegisterHotKey(wnd,hk_Id,MOD_ALT xor MOD_CONTROL,WORD('H'))) then
  MessageBox(0,'Hotkey registriert!',@szClassname[1],MB_OK or
MB_ICONINFORMATION);
```

RegisterHotKey-Definition

```
BOOL RegisterHotKey(
  HWND hWnd,          // handle to window
  int id,             // hot key identifier
  UINT fsModifiers,   // key-modifier options
  UINT vk             // virtual-key code
);
```

Hinweis

Beachten Sie bitte, dass der Rückgabewert **false** ist, wenn der Hotkey nicht registriert werden konnte. Das kann passieren, wenn Ihr Programm versucht, einen bereits registrierten Hotkey erneut zu registrieren. Gestalten Sie Ihr Programm daher möglichst so, dass der Hotkey nur der Unterstützung oder Ergänzung dient und keine vitale Funktion hat, von der der Ablauf des Programms abhängt.

4.3.1.2. Auf den Hotkey reagieren

Wenn der Anwender den Hotkey betätigt, löst das System die Nachricht "`WM_HOTKEY`" aus, die wir in unserem Programm bearbeiten. Im Parameter `wParam` steckt dabei die ID des Hotkeys, die wir bei der Registrierung festgelegt haben. Die Funktion lautet also ganz einfach nur:

```
WM_HOTKEY:
  if(wp = hk_Id) then
    MessageBox(0,'Sie haben den Hotkey ausprobiert',@szClassname[1],MB_OK);
```

Auf die gleiche Weise können Sie weitere Hotkeys abfragen.

Machen wir noch ein kleines Beispiel: Vielleicht kennen Sie die Funktion von Windows XP, die es Ihnen erlaubt, sich mit der Windows-Taste und L abzumelden. Ich bin nicht ganz sicher, ob Sie dazu die Powertoys installieren müssen oder ob die Funktion bereits eingebaut ist. Wir wollen sie für Windows 9x/ME einfach mal nachbilden.
Dazu müssen wir zuerst herausfinden, welches Betriebssystem aktiv ist. Wir benutzen deshalb "`GetVersionEx`". Diese Funktion erwartet von uns das `TOSVersionInfo`-Record, das wir vorher mit der Größenangabe initialisieren müssen:

```
wv.dwOSVersionInfoSize := sizeof(TOSVersionInfo);
GetVersionEx(wv);
```

TOSVersionInfo-Definition

```
typedef struct _OSVERSIONINFO{
  DWORD dwOSVersionInfoSize;        // Größe des Records
  DWORD dwMajorVersion;             // Versionsnr. vor dem Punkt
  DWORD dwMinorVersion;             // Versionsnr. nach dem Punkt
  DWORD dwBuildNumber;              // Build-Nr.
  DWORD dwPlatformId;               // Platform-ID
  TCHAR szCSDVersion[ 128 ];        // weitere Infos
} OSVERSIONINFO;
```

Von Interesse ist für uns der Wert von `dwPlatformId`, für den das System vordefinierte Konstanten bereithält. In unserem Fall heißt das:

```
if(wv.dwPlatformId = VER_PLATFORM_WIN32_WINDOWS) then
  RegisterHotKey(wnd,LogOff_Id,MOD_WIN,WORD('L'))
```

Angenommen, Sie wollen diesen Hotkey auch für Windows NT und 2000 registrieren, dann müssten zusätzlich die Konstante `VER_PLATFORM_WIN32_NT` und die Versionsnummer berücksichtigen:

```
WinNT := (wv.dwPlatformId = VER_PLATFORM_WIN32_NT) and
         (wv.dwMajorVersion <= 4);

Win2k := (wv.dwPlatformId = VER_PLATFORM_WIN32_NT) and
         (wv.dwMajorVersion = 5) and
         (wv.dwMinorVersion = 0); // XP = 5.1
```

Nachdem wir den Hotkey definiert haben, schreiben wir nun den Code für seine Aktion. Wie gesagt, er soll den Benutzer abmelden:

```
WM_HOTKEY:
  if(wp = LogOff_Id) then
    begin
      SendMessage(wnd,WM_CLOSE,0,0);
      ExitWindowsEx(EWX_LOGOFF,0);
    end;
```

4.3.1.3. Den Hotkey freigeben

Wenn Ihr Programm beendet wird, sollte es die registrierten Hotkeys auch wieder freigeben. Dazu benutzen Sie die API-Funktion "`UnregisterHotKey`", der Sie neben dem Fensterhandle wieder die ID des Hotkeys übergeben:

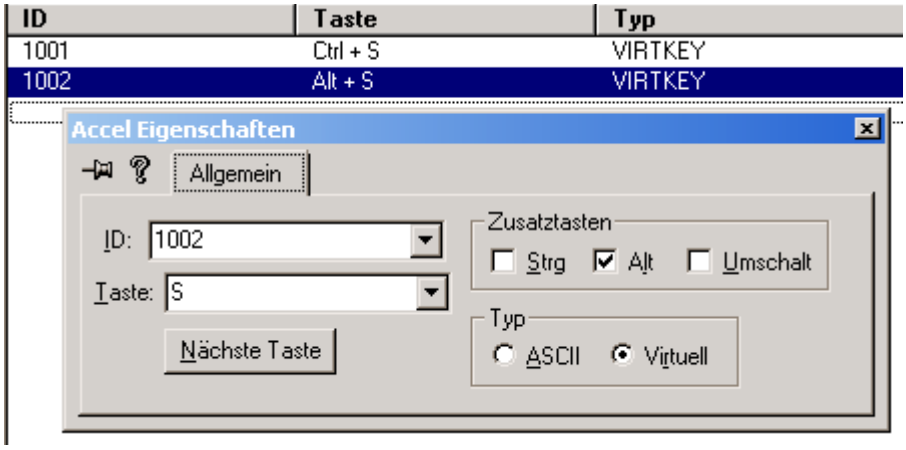
```
WM_DESTROY:
  begin
    UnRegisterHotKey(wnd,LogOff_Id);
    UnRegisterHotKey(wnd,hk_Id);

    PostQuitMessage(0);
  end;
```

4.3.2. Shortcuts

4.3.2.1. Shortcuts als Ressourcen definieren

Im Gegensatz zu den systemweiten Hotkeys sind Shortcuts anwendungsbezogen und werden in der gleichen Form oft in mehreren Anwendungen genutzt. Die bekanntesten Beispiele sind STRG+O für "Öffnen", STRG+S für "Speichern" usw.

Am einfachsten lässt sich ein Shortcut über eine Ressourcendatei festlegen. Der Vorteil ist, dass man seinen bevorzugten Editor benutzen kann. Ich habe für dieses Beispiel den Visual Studio 6-Editor verwendet, mit dem die Arbeit recht schnell und einfach geht:



Die Shortcut-Tabelle kann natürlich auch mit Hilfe eines Texteditors erstellt werden, wenn man das Format einer RC-Skriptdatei beachtet. In unserem Fall sehen die Definitionen so aus:

```
1000 ACCELERATORS DISCARDABLE
BEGIN
    "S",            1001,                   VIRTKEY, CONTROL
    "S",            1002,                   VIRTKEY, ALT
END
```

Ich habe hier zwei verschiedene Shortcuts definiert, STRG+S und ALT+S, die beide gebraucht werden. Nach dem Erzeugen der Ressourcendatei (*.res) muss diese noch in das Projekt eingefügt werden. Zur Nutzung der Shortcuts ziehen wir die Funktion "`LoadAccelerators`" heran, deren Rückgabewert das Handle der Shortcut-Tabelle ist. Diese Schritte sehen zusammengefasst so aus:

```
program accel;

{$R accel.res}

...

var
  hAccelTbl : DWORD;

...

begin
  { Tabelle mit Shortcuts laden }
  hAccelTbl := LoadAccelerators(hInstance, MAKEINTRESOURCE(1000));

  ...
end.
```

LoadAccelerators-Definition

```
HACCEL LoadAccelerators(
    HINSTANCE hInstance, // Modulinstanz
    LPCTSTR lpTableName  // ID der Shortcut-Tabelle
);
```

4.3.2.2. Shortcuts im Programm erzeugen

Man kann so eine Shortcut-Tabelle auch direkt im Programm erzeugen. Dazu benötigt man zuerst das `TAccel`-Record. Abhängig von der Anzahl der Shortcuts, die wir benötigen, definieren wir ein Array dieses Records und ordnen den jeweiligen Elementen dann die Daten zu. In unserem Fall haben wir nur zwei Shortcuts, so dass der programmtechnische Aufwand relativ gering ist:

```
var
  AccelTable : array[0..1]of TAccel;

begin
  // Shortcut #1 (STRG+S) definieren
  AccelTable[0].fVirt := FCONTROL or FVIRTKEY;
  AccelTable[0].key   := WORD('S');
  AccelTable[0].cmd   := IDC_ACCEL_SC;

  // Shortcut #2 (ALT+S) definieren
  AccelTable[1].fVirt := FALT or FVIRTKEY;
  AccelTable[1].key   := WORD('S');
  AccelTable[1].cmd   := IDC_ACCEL_CLOSE;
end.
```

Nun rufen wir nur die API-Funktion "`CreateAcceleratorTable`" auf, die als Parameter unser Array und die Anzahl der definierten Shortcuts erwartet:

```
hAccelTbl := CreateAcceleratorTable(AccelTable,2);
```

Hinweis

Die so erzeugte Shortcut-Tabelle muss vor dem Beenden des Programms mit der Funktion "`DestroyAcceleratorTable`" wieder freigegeben werden.

4.3.2.3. Auf den Shortcut reagieren

Hierfür ziehen wir die Nachricht "WM_COMMAND" heran, bei der im höherwertigen Wort (dem Benachrichtigungscode) von wParam eine Eins zu finden ist, wenn ein Shortcut aktiviert wurde. Die Abfrage ist daher:

```
WM_COMMAND:
  case HIWORD(wp) of
    1: { Eins = Shortcut aktiviert }
      case LOWORD(wp) of
        IDC_ACCEL_CLOSE:
          SendMessage(wnd, WM_CLOSE, 0, 0);
        IDC_ACCEL_SC:
          Messagebox(wnd, 'Strg+S gedrückt', 'Shortcut', MB_ICONINFORMATION);
      end;
  end;
```

4.3.2.4. Erweiterung der Nachrichtenschleife des Programms

Damit unser Programm tatsächlich auf die Shortcuts reagieren kann, ist die Funktion "TranslateAccelerator" notwendig, die wir in unsere Nachrichtenschleife einfügen. Diese Funktion übersetzt "WM_KEYDOWN"- und "WM_SYSKEYDOWN"-Nachrichten in "WM_COMMAND"- oder "WM_SYSCOMMAND"-Nachrichten und sendet diese dann an die Nachrichtenfunktion ("WndProc") unseres Fensters. Wenn die Funktion Null zurückliefert, dann sollte die Standard-Nachrichtenverarbeitung ausgeführt werden:

```
while(GetMessage(msg, 0, 0, 0)) do
  begin
    if(TranslateAccelerator(hWndMain, hAccelTbl, msg) = 0) then
      begin
        TranslateMessage(msg);
        DispatchMessage(msg);
      end;
  end;
```

TranslateAccelerator-Definition

```
int TranslateAccelerator(
    HWND hWnd,                // Fensterhandle
    HACCEL hAccTable,         // Handle der Shortcut-Tabelle
    LPMSG lpMsg               // Pointer des Nachrichten-Records
);
```

4.4. Datum und Uhrzeit

4.4.1. Vorwort

Dieses Tutorial will Ihnen kurz demonstrieren, wie man die Zeit und das Datum ausliest und bei der Darstellung die Systemeinstellungen des Anwenders berücksichtigt. Letzteres ist natürlich bei Dateimanagern u.ä. sinnvoll. Aber selbst wenn Sie "nur" ein Programm schreiben, das irgendwo die aktuelle Zeit und das aktuelle Datum einblendet, sollten Sie die Einstellungen des Anwenders nicht ignorieren. In einigen Programmen sieht man beim Datum z.B. fest eingestellte Formatierungen, etwa

```
30.11.2002
```

oder auch

```
11/30/2002
```

Das mag Ihnen gefallen. Der Anwender muss aber nicht ebenso denken. Prüfen Sie doch einfach mal, ob Ihre eigenen Ländereinstellungen mit Ihrem Geschmack, was die Darstellung von Datum und Zeit angeht, übereinstimmen. Die Standardeinstellung von Windows 98 wäre beispielsweise:

```
TT.MM.JJ
```

Windows 2000 und XP benutzen das selbe Format, allerdings mit einer vierstelligen Jahreszahl. Einige Anwender verwenden aber auch eine ISO-8601-konforme Anzeige, wie etwa

```
JJJJ-MM-TT
```

Solche benutzerdefinierten Einstellungen sollten Sie nach Möglichkeit nicht ignorieren, wenn Sie nicht den guten Eindruck zerstören wollen, den Sie mit Ihrem Programm sicher machen möchten.

4.4.2. Datum und Uhrzeit auslesen

Bevor wir uns um die Darstellung von Datum und Uhrzeit kümmern können, müssen wir diese Informationen zuerst einmal in Erfahrung bringen. Das Betriebssystem bietet dazu die beiden Funktionen "`GetLocalTime`" und "`GetSystemTime`". "`GetLocalTime`" liefert dabei die lokale Zeit und das lokale Datum zurück.

```
GetLocalTime(lSysTime);
```

"`GetSystemTime`" hingegen liefert Zeit und Datum im UTC-Format.

```
GetSystemTime(lSysTime);
```

Die UTC (koordinierte Weltzeit) ist die einheitliche Grundlage für die Zeitbestimmung im täglichen Leben; auch im internationalen Bereich. Man muss nur wissen, wie sich die jeweilige lokale Zeit zur UTC verhält. In Deutschland entspricht die lokale Zeit z.B. UTC + 1 Stunde (zur Sommerzeit UTC + 2 Stunden).

Beiden Prozeduren wird eine Variable vom Typ `TSystemTime` übergeben.

SYSTEMTIME-Definition

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

4.4.3. Die GetTimeFormat-Funktion

Die Funktion "GetTimeFormat" formatiert eine Zeitangabe, so dass sie dem lokalen "Zeitstring" entspricht. Dabei ist es egal, ob Sie eine UTC- oder eine lokale Zeitangabe verwenden.
Der erste Parameter gibt an, welches Format herangezogen werden soll. Zwei Konstanten (LOCALE_SYSTEM_DEFAULT,LOCALE_USER_DEFAULT) sind definiert:

```
GetTimeFormat(LOCALE_USER_DEFAULT,
```

Der zweite Parameter ist eine Kombination von Flags. Im Beispiel habe ich mich für

```
  TIME_FORCE24HOURFORMAT,
```

entschieden, damit das 24-Stunden-Format in jedem Fall benutzt wird. Wenn Sie z.B. auf die Anzeige der Sekunden verzichten wollen, dann benutzen Sie zusätzlich TIME_NOSECONDS:

```
  TIME_FORCE24HOURFORMAT or TIME_NOSECONDS
```

Weitere Angaben finden Sie im MSDN und PSDK.

Der dritte Parameter ist ein Zeiger auf die TSystemTime-Variable, die die ausgelesene Zeit enthält

```
  @st,
```

Wenn Sie diesen Parameter auf **nil** setzen, wird die lokale Zeit verwendet. In den meisten Fällen dürfte dies ausreichen. Da wir im Beispielprogramm aber auch die UTC-Zeit anzeigen wollen, müssen wir hier die Variable benutzen.

Der vierte Parameter ist eigentlich ein Formatstring, mit dem Sie die Darstellung der Zeit beeinflussen können. Da wir aber den Formatstring des Systems nutzen wollen, kann dieser Parameter leer bleiben

```
  nil,
```

Die beiden letzten Parameter spezifizieren ein Char-Array und dessen Länge:

```
  buf,
  sizeof(buf))
```

Im Fehlerfall liefert die Funktion den Wert Null zurück, und Sie können mit "GetLastError" die Ursache dafür herausfinden:

```
if(GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  @st,nil,lpBuf,sizeof(buf)) = 0) then
case GetLastError of
  ERROR_INSUFFICIENT_BUFFER:
    MessageBox(0,'Nicht ausreichender Puffer',nil,0);
  ERROR_INVALID_FLAGS:
    MessageBox(0,'Ungültige Flags',nil,0);
  ERROR_INVALID_PARAMETER:
    MessageBox(0,'Ungültige Parameter',nil,0);
  { ... }
end;
```

Im Erfolgsfall wird der übergebende Puffer (im Beispiel "`lpBuf`") mit dem formatierten Zeitstring gefüllt, und die Funktion liefert die Anzahl der kopierten Zeichen zurück (einschließlich des Zeichens `#0`). Dies können Sie z.B. bei dynamischen Array nutzen, bei denen Sie die Größe erst zur Laufzeit festlegen. Aber ein Array wie:

```
var
  buf : array[0..MAX_PATH]of char;
```

reicht in den meisten Fällen aus.

GetTimeFormat-Definition

```
int GetTimeFormat(
  LCID Locale,              // locale-Info
  DWORD dwFlags,            // Flags
  CONST SYSTEMTIME *lpTime, // Zeiger auf "TSystemTime"-Variable
  LPCTSTR lpFormat,         // Formatstring
  LPTSTR lpTimeStr,         // Puffer für die Ausgabe
  int cchTime               // Größe des Puffers
);
```

4.4.3.1. Formatstring selbst definieren

Wenn Sie die Formatierung dennoch einmal selbst übernehmen wollen, dann definieren Sie einen Formatstring wie beispielsweise:

```
GetTimeFormat(0,
  TIME_FORCE24HOURFORMAT,
  @st,
  'hh mm ss' { <-- },
  buf,
  sizeof(buf))
```

In diesem Beispiel würde die Uhrzeit so angezeigt werden:

```
16 50 10
```

Symbol	Bedeutung
h	einstellige Stundenanzeige (12-Stunden-Format)
hh	zweistellige Stundenanzeige (12-Stunden-Format)
H	einstellige Stundenanzeige (24-Stunden-Format)
HH	zweistellige Stundenanzeige (24-Stunden-Format)
m	einstellige Minutenanzeige
mm	zweistellige Minutenanzeige
s	einstellige Sekundenanzeige
ss	zweistellige Sekundenanzeige
t	einstelliger "Time-Marker"; A oder P
tt	zweistelliger "Time-Marker"; AM oder PM

4.4.4. Die GetDateFormat-Funktion

Zum Formatieren des Datums verwenden wir die Funktion "`GetDateFormat`". Wie auch bei "`GetTimeFormat`" bestimmt der erste Parameter das Format:

```
GetDateFormat(LOCALE_USER_DEFAULT,
```

Der zweite Parameter definiert wieder Flags. Dieser Wert muss allerdings Null sein, wenn der Formatstring (dazu gleich) benutzt wird. Im Beispiel habe ich mich für

```
  DATE_SHORTDATE,
```

entschieden, was dem kurzen Datumsformat aus den Ländereinstellungen entspricht. Es ließe sich auch `DATE_LONGDATE` benutzen, was logischerweise dem langen Datumsformat entspricht. MSDN und PSDK enthalten weitere Parameter, wobei zu beachten ist, dass sich einige gegenseitig ausschließen und nicht gemeinsam verwendet werden sollten.

Der dritte Parameter ist auch hier ein Zeiger auf die `TSystemTime`-Variable, die das Datum enthält. Wenn Sie ihn auf **nil** setzen, wird das aktuelle Datum herangezogen

```
  nil
```

Der Formatstring an vierter Stelle bleibt in unserem Fall wieder leer, da wir ja die Formatierung des Systems nutzen wollen

```
  nil,
```

An letzter Stelle stehen ebenfalls wieder ein Puffer (Char-Array) und dessen Länge

```
  buf,
  sizeof(buf))
```

Im Fehlerfall liefert die Funktion ebenfalls Null zurück, und über "`GetLastError`" lässt sich wieder der Grund dafür herausfinden. Im Erfolgsfall wird der übergebene Puffer mit dem formatierten Datumsstring gefüllt, und die Funktion gibt die Anzahl der kopierten Zeichen (inkl. `#0`) zurück.

Wenn Sie auch hierbei ein dynamisches Array nutzen wollen, gilt das gleiche wie bei der Funktion "`GetTimeFormat`".

GetDateFormat-Definition

```
int GetDateFormat(
  LCID Locale,               // locale-Info
  DWORD dwFlags,             // Flags
  CONST SYSTEMTIME *lpDate,  // Zeiger auf "TSystemTime"-Variable
  LPCTSTR lpFormat,          // Formatstrings
  LPTSTR lpDateStr,          // Puffer für die Ausgabe
  int cchDate                // Größe des Puffers
);
```

4.4.4.1. Formatstring selbst definieren

Auch hier lässt sich die Systemeinstellung durch einen eigenen Formatstring umgehen. Dazu müssen Sie allerdings zuerst die Flags auf Null setzen, dann können Sie die eigene Formatierung verwenden:

```
GetDateFormat(LOCALE_USER_DEFAULT,
  0,                    // Flags
  nil,
  'ddd'',' MMM dd yy', // Formatstring
  buf,
  sizeof(buf))
```

Das Ergebnis wäre in diesem Fall:

```
Sa, Nov 30 02
```

Symbol	Bedeutung
d	einstellige Tagesanzeige
dd	zweistellige Tagesanzeige
ddd	der Name des Tages wird in Kurzform angezeigt, etwa "Mo" für Montag, usw.
dddd	der Name des Tages wird angezeigt
M	einstellige Monatsanzeige
MM	zweistellige Monatsanzeige
MMM	der Monatsname wird in Kurzform angezeigt, etwa "Jan" für Januar, usw.
MMMM	der Name des Monats wird angezeigt
y	einstellige Jahresanzeige
yy	zweistellige Jahresanzeige
yyyy	vierstellige Jahresanzeige

4.4.4.2. Mit dynamischen Puffern arbeiten

Der Vollständigkeit halber soll noch kurz diese Variante demonstriert werden. Zuerst benötigen Sie eine Variable vom Typ `pchar`. Da Sie anfangs nicht wissen, wie viele Zeichen benötigt werden, müssen Sie genau das zuerst herausfinden:

```
iLen := GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  @st,nil,nil,0);
```

"`iLen`" ist dabei eine `integer`-Variable. Da die Funktion die Anzahl der benötigten Zeichen zurückliefert, können wir abhängig von diesem Ergebnis Speicher für unsere `pchar`-Variable reservieren:

```
GetMem(lpBuf,iLen);
```

Die Funktion "GetMem" löst eine Exception aus, wenn kein freier Speicher zur Verfügung steht. Die weiteren Anweisungen sollten daher in einen **try**/**finally**-Block gekapselt werden. Konnte der Speicher reserviert werden, dann rufen Sie die Funktion erneut auf und geben diesmal natürlich die Variable an:

```
ZeroMemory(lpBuf,iLen);

if(GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  @st,nil,lpBuf,iLen) = iLen) then
SetWindowText(wnd,lpBuf);
```

In diesem Fall wird die Anzeige der (formatierten) Uhrzeit sogar noch davon abhängig gemacht, ob das Funktionsergebnis der Anzahl der Zeichen entspricht. Am Schluss geben Sie die Variable wieder frei:

```
FreeMem(lpBuf,iLen);
```

Das Beispielprogramm enthält beide Versionen, die Sie mit Hilfe eines Compilerschalters auswählen können.

Ein kleines Problem, auf das ich aufmerksam gemacht wurde, soll nicht verschwiegen werden. Gehen wir dabei von dem Fall aus, dass Sie die Funktion ("GetTimeFormat" in dem Fall) zweimal verwenden: einmal bestimmen Sie die Größe des Puffers, beim zweiten Mal lesen Sie die Zeit aus und erhalten Ihren formatierten Zeitstring.

Beispiel: Sie bestimmen um 9 Uhr 30 und 17 Sekunden die Größe des Puffers. Kurz darauf (abhängig von Ihrer Systemauslastung wäre es denkbar, dass es mittlerweile 9 Uhr 30 und 18 ... Sekunden ist) lesen Sie die Zeit aus. In beiden Fällen ist der Puffer meiner Meinung nach ausreichend dimensioniert, so dass es zu keinem Problem kommen sollte.
Anders sieht es dagegen um 9 Uhr 59 und 59 Sekunden aus. Hier bestimmen Sie die Größe des Puffers. Ausgehend von den selben Verzögerungen im System wäre es nun denkbar, dass die Zeit tatsächlich erst um 10 Uhr und X Sekunden ausgelesen wird. In diesem Fall könnte der zuvor bestimmte Puffer bereits wieder zu klein sein. Die Funktion würde als Ergebnis Null zurückliefern, und der Grund wäre (via "GetLastError") der Fehler ERROR_INSUFFICIENT_BUFFER.

Dieses Problem könnte sich evtl. mit folgendem Code provozieren lassen:

```
iLen := GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  nil,nil,nil,0);

{ ... }

GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  nil,nil,lpBuffer,iLen);
```

Zuerst wird die Puffergröße bestimmt, und da der Zeitparameter **nil** ist, wird die aktuelle Zeit herangezogen. Nachdem der Speicher reserviert wurde, findet ein weiterer Zugriff statt. Diesmal mit der Puffervariablen und der vorher ermittelten Puffergröße. Und da auch hier der Zeitparameter wieder **nil** ist, wird erneut die aktuelle Zeit als Grundlage benutzt. (Sie dürfen auch gern absichtlich eine Verzögerung mit "sleep" einbauen, wenn Sie es "mal krachen lassen wollen ..." :o))

Im Fall des Beispielprogramms sollte aber dennoch (zumindest bei der Formatierung der Zeit) nichts passieren. Zum einen wird eine Funktion im Programm aufgerufen, der u.a. eine TSystemTime-Variable übergeben wird. Diese wird dann sowohl zur Bestimmung der Puffergröße als auch zur eigentlichen Formatierung herangezogen. Solange Sie also nicht innerhalb der Funktion die Zeit ein zweites Mal bestimmen lassen, tritt das geschilderte Problem nicht auf.
Außerdem wird, ich erwähnte es ja bereits, die Anzeige der Zeit noch davon abhängig gemacht, ob das Funktionsergebnis mit der zuvor bestimmten Puffergröße identisch ist:

```
if(GetTimeFormat(LOCALE_USER_DEFAULT,TIME_FORCE24HOURFORMAT,
  @st,nil,lpBuf,iLen) = iLen) then SetWindowText(wnd,lpBuf);
```

Das Problem könnte also bestenfalls nur bei der Formatierung des Datums auftreten, da hier der jeweils aktuelle Wert verwendet wird. Aber auch dann dürfte nichts passieren, da die gleiche Absicherung getroffen wurde:

```
if(GetDateFormat(LOCALE_USER_DEFAULT,DATE_SHORTDATE,
  nil,nil,lpBuf,iLen) = iLen) then SetWindowText(wnd, lpBuf);
```

Schlimmstenfalls bleibt also das jeweilige Feld für eine Sekunde leer.

4.4.5. Datum und Uhrzeit ändern

Auch wenn es im Beispielprogramm keine Rolle spielt; erwähnenswert ist das Thema sicher. Datum und Uhrzeit kann man mit den beiden Funktionen "`SetSystemTime`" (UTC-Zeit) und "`SetLocalTime`" ändern. Beide Funktionen erwarten als Parameter eine `TSystemTime`-Variable, die man vorher mit den neuen Werten "füttern" sollte. Beispielsweise

```
var
  st : TSystemTime;
begin
  st.wYear         := 2003;
  st.wMonth        := 1;
  st.wDay          := 1;
  st.wHour         := 1;
  st.wMinute       := 0;
  st.wSecond       := 0;
  st.wMilliseconds := 0;

  if(SetLocalTime(st)) then
    MessageBox(0,'Zeit geändert','SetLocalTime',0);
end;
```

SetLocalTime-Definition

```
BOOL SetLocalTime(
  const SYSTEMTIME* lpSystemTime
);
```

SetSystemTime-Definition

```
BOOL SetSystemTime(
  const SYSTEMTIME* lpSystemTime
);
```

Die Rechtefrage unter Windows NT/2000/XP

Die Rechtefrage ist der eigentliche Grund dieses Kapitels. Wenn man sich die alte "`win32.hlp`" ansieht, die Delphi beiliegt, dann steht dort folgendes:

Win32 Programmer´s Reference

The **SetSystemTime** function fails if the calling process does not have the SE_SYSTEMTIME_NAME privilege. This privilege is disabled by default. Use the AdjustTokenPrivileges function to enable this privilege and again to disable it after the time has been set.

Sinngemäß also: "Wenn der aufrufende Prozess (das Programm z.B.) nicht das Recht `SE_SYSTEMTIME_NAME` besitzt, scheitert der Aufruf der Funktion und damit das Ändern von Datum und Uhrzeit. Es wird empfohlen, das Recht zu aktivieren, die Funktion auszuführen und das Recht dann wieder zu deaktivieren." Vergleicht man diese Aussage mit neueren Ausgaben des Platform SDK, so liest man:

Platform SDK

The **SetSystemTime** function enables the SE_SYSTEMTIME_NAME privilege before changing the system time. This privilege is disabled by default.

Hier heißt es zwar auch, dass das Recht `SE_SYSTEMTIME_NAME` standardmäßig deaktiviert ist. Es steht aber auch da, dass die Funktion dieses Recht wohl selbst holen kann (*enables the ... privilege*). Das steht im Widerspruch zur Aussage der alten Hilfedatei. Ich bin allerdings geneigt, den neuen Quellen eher zu glauben. Ein Test unter Windows XP ergab nämlich einen ganz anderen Ansatzpunkt.

Doch zuerst führen wir den Ratschlag der alten Hilfedatei aus. Auf der Grundlage eines Beitrags, den ich einem Forum entnommen habe, schrieb ich folgende kleine Funktion, die gezielt Rechte für das aufrufende Programm aktivieren und deaktivieren kann:

```
function SetPrivilege(pszPrivilege: LPCSTR;
  bEnablePrivilege: boolean): boolean;
var
  wv    : TOSVersionInfo;
  Token : THandle;
  tkp   : TTokenPrivileges;
begin
  wv.dwOSVersionInfoSize := sizeof(TOSVersionInfo);
  GetVersionEx(wv);

  // da unter 9x jeder alles darf, setzen wir das
  // Ergebnis in dem Fall auf TRUE
  Result := (wv.dwPlatformId = VER_PLATFORM_WIN32_WINDOWS);

  // jetzt der echte Code für NT-Plattformen
  if(wv.dwPlatformId = VER_PLATFORM_WIN32_NT) then begin
    if(not OpenProcessToken(GetCurrentProcess,
      TOKEN_ADJUST_PRIVILEGES or TOKEN_QUERY,Token)) then exit;

    try
      if(not LookupPrivilegeValue(nil, pszPrivilege,
        tkp.Privileges[0].Luid)) then Exit;

      tkp.PrivilegeCount := 1;

      if(bEnablePrivilege) then
        tkp.Privileges[0].Attributes := SE_PRIVILEGE_ENABLED
      else
        tkp.Privileges[0].Attributes := 0;

      Result := AdjustTokenPrivileges(Token, false, tkp,
        0, PTokenPrivileges(nil)^, PDWord(nil)^);
    finally
      CloseHandle(Token);
    end;
  end;
end;
```

Bevor wir also die Zeit und/oder das Datum ändern können, müssen wir uns das dazu erforderliche Recht `SE_SYSTEMTIME_NAME` holen, was im Code so aussieht:

```
const
  SE_SYSTEMTIME_NAME = 'SeSystemtimePrivilege';

{ ... }

if(SetPrivilege(SE_SYSTEMTIME_NAME,true)) then
  SetLocalTime(st);
```

So würde das auch funktionieren, wenn ... tja ...

Sagen wir so: bei einem Test mit Benutzerrechten wurde die Zeit auch mit der eben erfolgten Anweisung nicht geändert.

Des Rätsels Lösung war eine Einstellung der lokalen Sicherheitsrichtlinien von Windows XP. Hier finden Sie unter "Lokale Richtlinien/Zuweisen von Benutzerrechten" den Punkt "Ändern der Systemzeit", für den standardmäßig nur die Administratoren und Hauptbenutzer eingetragen sind. Geben Sie hier zusätzlich Ihren Benutzernamen an, oder ergänzen Sie die Gruppe, der Sie als Benutzer angehören, dann können Sie auch Datum und Uhrzeit ändern.

Und das klappt dann auch ohne das `SE_SYSTEMTIME_NAME`-Recht! Zumindest war das bei meinem Test unter Windows XP so; und da das PSDK keinerlei Einschränkungen nennt, dürfte es bei Windows 2000 und NT ähnlich sein.

4.4.6. Ausflug ins Internet

Als kleine Ergänzung zum Thema wollen wir uns noch kurz ansehen, wie wir die Zeit von einem Server im Internet holen können. Als neue Units benötigt unser Programm dafür die "WinSock.pas" und die "WinInet.pas". Letztere es wichtig, weil das Beispielprogramm kein Dialer ist! Es stellt keine eigene Verbindung her, sondern es wartet, bis eine Verbindung aktiv ist und gibt erst dann die entsprechenden Funktionen frei.

Ein besonderer Dank gebührt Peter Gaede. Er lieferte nicht nur die Idee für dieses Kapitel, er löste auch das "10-Mio-Rätsel", mit dem ich erst nichts anfangen konnte. :o)

4.4.6.1. WinSock initialisieren

Die erste Funktion, die wir aufrufen müssen, heißt "WSAStartup". Ohne diese Initialisierung können wir keine Netzwerkfunktionen verwenden. Bitte verwechseln Sie das nicht mit Internet-Funktionen! Netzwerkfunktionen, die uns im Rahmen dieses Kapitels interessieren werden, sind etwa

```
socket
connect
recv
gethostbyname
```

Doch zurück zu "WSAStartup" -
In unserem Fall genügt es, die Funktion einmal beim Start aufzurufen. Sie erwartet als ersten Parameter die gewünschte WinSock-Version. Ich habe mich für v1.1 entschieden, was bei den bisher bekannten Windows-Versionen eigentlich keine Probleme verursachen sollte. Im Zweifelsfall (Win95?) können Sie aber auch die Version 1.0 benutzen. Der zweite Parameter der Funktion ist eine TWsaData-Variable:

```
if(WsaStartup(MAKEWORD(1,1),wsadata) <> 0) then begin
  MessageBox(0,'WinSock-Fehler',nil,0);
  Halt;
end;
```

Die Funktion liefert im Erfolgsfall Null zurück. Tritt ein Fehler auf, ist das Rückgabeergebnis der entsprechende Fehlercode. In dem Fall sollten Sie das Programm besser beenden oder wenigstens die Netzwerkfunktionen deaktivieren.

Wenn Sie keine Netzwerkfunktionen mehr benötigen (in unserem Fall beim Beenden des Programms), rufen Sie bitte "WSACleanup" auf.

```
WsaCleanup;
```

4.4.6.2. Bin ich schon drin?

Woher wissen wir nun, wann wir online sind? Dazu stellt uns die Unit "WinInet.pas" die Funktion "InternetGetConnectedState" zur Verfügung. Diese Funktion besitzt nur zwei Parameter, von denen wir den ersten ignorieren können. Der zweite ist reserviert und interessiert daher auch nicht. Die Funktion liefert **true** zurück, wenn wir online sind:

```
Result := (InternetGetConnectedState(nil,0));
```

Das Beispielprogramm enthält aber noch eine erweiterte Funktionalität. Im Normalfall prüfen wir nur, ob überhaupt eine Internetverbindung besteht. (Wie gesagt: das Programm wählt sich ja nicht selbst ein!) Um uns später mit dem Time-Server verbinden zu können, benötigen wir dessen IP-Adresse. Dazu nutzen wir die Funktion "GetHostByName", die aber nur im Erfolgsfall (sprich: bei aktiver Web-Verbindung) aufgerufen wird. Beispielsweise:

```
hostent := gethostbyname('ntpserv.fh-magdeburg.de');
```

Die Funktion "GetHostByName" holt Informationen zu der angegebenen URL und gibt diese, als Funktionsergebnis, an eine Variable vom Typ PHostEnt weiter. Im Fall eines Fehlers bzw. wenn wir nicht (mehr) online sind, ist diese Variable **nil**.

All das passiert im Beispiel in einer eigenen Funktion, die Sie wie folgt aufrufen:

```
if(not IsOnline('ntpserv.fh-magdeburg.de',@dwIp)) or
  (dwIp = 0) then exit;
```

Falls Sie online sind und der benutzte Time-Server erreichbar war, enthält die Variable im zweiten Parameter ("dwIp" im Beispiel) die IP-Adresse des Servers.

Wenn Sie lediglich testen wollen, ob Sie online sind, genügt auch der Aufruf

```
if(not IsOnline(nil,nil)) then { ... }
```

der auch schneller ist, weil keine URL aufgelöst werden muss. Es ist ebenfalls möglich, nur eine URL anzugeben:

```
if(not IsOnline('www.microsoft.de',nil)) then { ... }
```

In diesem Fall wird das Funktionsergebnis (online oder nicht online) davon abhängig gemacht, dass a) eine aktive Web-Verbindung besteht und b) die benutzte URL erreichbar ist. Ist das nicht der Fall, dann wäre das Ergebnis **false**, obwohl Sie evtl. eine aktive Web-Verbindung haben und damit ja eigentlich online sind.
Die IP wird in beiden Fällen allerdings ignoriert - weil: nicht benutzt.

Hinweis

Die IP-Adresse wird in der so genannten "host byte order" zurückgegeben. Das ist für unsere Zwecke in Ordnung, aber wenn Sie die IP z.B. in Ihrem Programm anzeigen wollen, müssen Sie sie vorher "drehen". Warum? Nehmen wir als Beispiel den localhost. Würden Sie seine IP wie folgt bestimmen:

```
hostent := gethostbyname('localhost');
dwIp    := integer(POINTER(hostent^.h_addr_list^)^);
```

dann wäre das Ergebnis "1.0.0.127" (formatiert als IP-Adresse). Nun sehen Sie selbst, dass dieser Wert einfach nur "verdreht" ist. Mit der Funktion "htonl" konvertieren Sie ihn in die "TCP/IP network byte order", so dass die erwartete Adresse "127.0.0.1" daraus wird.

4.4.6.3. Die Zeit vom Server holen

Zum Verbinden mit dem Server erstellen wir zunächst ein Socket:

```
s := socket(AF_INET,SOCK_STREAM,0);
```

Der erste Parameter ist die Angabe der Protokollfamilie. AF_INET schließt hierbei TCP, UDP u.ä. ein. Der zweite Parameter definiert die Art der Verbindung, wobei SOCK_STREAM hier für eine TCP-Verbindung steht.
Im Fehlerfall ist das Ergebnis der Funktion INVALID_SOCKET, wobei sich dann mit "WSAGetLastError" die genaue Fehlerursache herausfinden lässt.

Konnte das Socket erstellt werden, benötigen wir eine TSockAddr-Variable, der wie den Port und die IP-Adresse des Servers übergeben. Und damit sind wir schon beim "time protocol":

Üblicherweise verbinden wir uns mit dem Server auf Port 37. Ist das geschehen, liefert uns der Server einen Zeitwert (32 Bit) zurück und beendet die Verbindung. Der übermittelte Wert entspricht der Anzahl der Sekunden seit dem 1. Januar 1900, Null Uhr.

Wie gehen wir nun vor?
Zunächst die schon erwähnte TSockAddr-Variable:

```
saddr.sin_family      := AF_INET;
saddr.sin_addr.S_addr := dwIp;      // IP-Adresse des Servers (host byte order!)
saddr.sin_port        := htons(37);
```

Im nächsten Schritt verbinden wir uns mit dem Server. Dazu dient die Funktion "`connect`", die neben dem Socket auch die eben gefüllte `TSockAddr`-Variable und ihre Größe als Parameter erwartet:

```
res := connect(s,saddr,sizeof(TSockAddr));
```

Im Erfolgsfall lesen wir dann mit Hilfe der Funktion "`recv`" den Zeitwert in eine `dword`-Variable. Die Funktion erwartet natürlich wieder die Socket-Variable, gefolgt von der `dword`-Variable für den Zeitwert und ihrer Größe. Die Flags im letzten Parameter setzen wir auf Null:

```
if(res <> SOCKET_ERROR) then
  res := recv(s,iTime,sizeof(iTime),0);
```

Der ausgelesene Wert liegt nun aber in der "host byte order" vor. Bevor wir ihn verwenden können, müssen wir ihn mit Hilfe von "`htonl`" "drehen":

```
iTime := htonl(iTime);
```

Sie können diesen Schritt gern einmal auslassen. ;o)

Es wurde bereits gesagt, dass sich der Zeitwert auf den 1. Januar 1900, Null Uhr, bezieht. Wir müssen also dieses Datum zu dem Wert addieren, den wir vom Server bekommen haben. Dazu benutzen wir zunächst eine `TSystemTime`-Variable, die wir entsprechend einstellen

```
st.wYear            := 1900;
st.wMonth           := 1;
st.wDayOfWeek       := 0;
st.wDay             := 1;
st.wHour            := 0;
st.wMinute          := 0;
st.wSecond          := 0;
st.wMilliseconds    := 0;
```

und mit der Funktion "`SystemTimeToFileTime`" in einen `FILETIME`-Wert umrechnen

```
if(not SystemTimeToFileTime(st,ft)) then goto SocketClose;
```

Nun können wir die beides addieren. Der Zeitwert vom Server muss allerdings als `int64` übergeben und mit 10.000.000 (10 Millionen) multipliziert werden:

```
li.QuadPart         := (int64(iTime) * 10000000) +
  ULARGE_INTEGER(ft).QuadPart;
```

Warum?
`FILETIME` ist ein 64-Bit-Integer, der die Anzahl von 100ns-Intervallen (Nanosekunden) pro Sekunde seit dem 1. Januar 1601 wiedergibt (UTC). Der 1. Januar 1900, den wir mit "`SystemTimeToFileTime`" umgerechnet haben, enthält bereits die Anzahl dieser Intervalle. Nur der Server-Wert muss noch entsprechend ergänzt werden.

Und der Wert von 10.000.000 ergibt sich, weil pro Sekunde eben 10 Millionen solcher 100ns-Intervalle möglich sind:

```
1sec = 1.000 msec = 1.000.000 µsec = 1.000.000.000 ns
```

Der so ermittelte Wert kann nun mit der Funktion "`FileTimeToSystemTime`" wieder in eine `TSystemTime`-Variable umgewandelt werden und sollte dann eigentlich die gültige Zeit und das gültige Datum enthalten.

```
if(not FileTimeToSystemTime(TFileTime(li),st)) then goto SocketClose;
```

SystemTimeToFileTime-Definition

```
BOOL SystemTimeToFileTime(
  const SYSTEMTIME* lpSystemTime, // TSystemTime-Konstante
  LPFILETIME lpFileTime           // TFileTime-Variable
);
```

FileTimeToSystemTime-Definition

```
BOOL FileTimeToSystemTime(
  const FILETIME* lpFileTime, // TFileTime-Konstante
  LPSYSTEMTIME lpSystemTime   // TSystemTime-Variable
);
```

4.5. Die Verwendung von INI-Dateien

4.5.1. Vorwort

Dieser kleine Beitrag befasst sich mit den wichtigsten Funktionen bezüglich INI-Dateien. Auch wenn Microsoft die Nutzung der Registry vorschlägt, spricht nichts gegen die Verwendung dieser Dateien zur Speicherung von Konfigurationsdaten.

Das Hauptaugenmerk liegt allerdings auf den "privaten" Funktionen (ich nenne sie einmal so), die das Laden und Speichern von Daten in eigenen Dateien und Ordnern erlauben. Um die Funktionen, die auf die "`win.ini`" zugreifen, wollen wir einen Bogen machen, denn unsere Konfigurationsdaten sollten nach Möglichkeit immer in eigenen Dateien gespeichert werden.

4.5.2. Werte aus der INI-Datei lesen

Gehen wir im Beispiel von folgender INI-Datei aus, die ein paar Benutzerangaben enthält *(Ähnlichkeiten mit lebenden Personen sind rein zufällig!)*:

```
[names]
Paul=Paul

[Paul]
location=Salzburg
age=45
```

Wie Sie sehen können, befindet sich in der INI-Datei eine Sektion `names`, in der zweimal der Name "Paul" aufgeführt ist. Der Hintergedanke dabei wird zu einem späteren Zeitpunkt erklärt werden. Weiterhin sehen Sie, dass der Name "Paul" noch einmal als eigene Sektion erscheint und dass dort Wohnort und Alter eingetragen sind.

Wir wollen nun folgendes machen: Wir gehen davon aus, dass uns der gesuchte Name bekannt ist - "Paul" in diesem Fall. Also öffnen wir direkt die gleichnamige Sektion und lesen Wohnort und Alter aus. Dazu steht uns für Strings die Funktion "`GetPrivateProfileString`" zur Verfügung. Es wäre natürlich auch möglich, damit numerische Werte zu lesen, aber das wäre nicht im Sinne dieses Beitrags. Daher lesen wir das Alter mit der Funktion "`GetPrivateProfileInt`".

4.5.2.1. GetPrivateProfileString

Der erste Parameter der Funktion "`GetPrivateProfileString`" ist der Name der Sektion, aus der wir einen String auslesen wollen:

```
GetPrivateProfileString('Paul',
```

Der nächste Parameter ist der Name des Topics, der auf den gesuchten String verweist

```
  'location',
```

Es folgt ein Vorgabewert, der im Fehlerfall benutzt wird. Kann der gesuchte Topic nicht gefunden werden, wird stattdessen dieser Wert zurückgeliefert

```
  nil,
```

Nun folgt der Textpuffer, der den gesuchten String aufnehmen soll, und dessen Größe:

```
  location,
  BUFSIZE,
```

Und zu guter Letzt muss der Name der INI-Datei angegeben werden

```
  pchar(ExtractFilePath(paramstr(0)) + szIniFile));
```

Der Rückgabewert der Funktion ist die Anzahl der kopierten Zeichen. Das ist sehr hilfreich, da Sie so auch mit String-Variablen arbeiten können; beispielsweise:

```
SetLength(szBuffer,MAX_PATH);
SetLength(szBuffer,GetPrivateProfileString('Paul',
  'location',nil,@szBuffer[1],MAX_PATH,'userdata.ini'));
```

In unserem Fall interessiert uns der Rückgabewert allerdings nicht, da der Puffer entweder den Wohnort enthält oder (im Fehlerfall) auf den Vorgabewert **nil** gesetzt wird. Mit anderen Worten: entweder wird der Wohnort der ausgewählten Person angezeigt, oder das Feld bleibt leer.

4.5.2.2. GetPrivateProfileInt

Das Prinzip zum Auslesen von numerischen Werten ähnelt dem der Strings. Es entfällt eigentlich nur der Textpuffer und dessen Größe. Und der Vorgabewert ist hier natürlich auch ein numerischer Typ:

```
age := GetPrivateProfileInt('Paul',                  // Sektionsname
  'age',                                             // Topic
  -1,                                                // Vorgabewert
  pchar(ExtractFilePath(paramstr(0)) + szIniFile));  // Datei
```

Allerdings spielt diesmal der Rückgabewert eine sehr große Rolle, denn er ist der ausgelesene Wert, bzw. im Fehlerfall unser Vorgabewert.

4.5.3. Werte in die INI-Datei eintragen

Natürlich enthält das Beispielprogramm auch die Möglichkeit, selbst Namen in die INI-Datei einzutragen. Dazu stehen Ihnen drei Eingabefelder zur Verfügung, die Sie entsprechend füllen können.

Anzumerken ist, dass es keine spezielle Funktion zum Speichern von numerischen Werten gibt. In unserem Fall ist das nicht weiter tragisch, da das Alter aus dem Eingabefeld ohnehin als Text (String) ausgelesen wird. Sollten Sie in Ihren Anwendungen jedoch mit echten numerischen Werten arbeiten, müssen Sie diese zunächst in einen String konvertieren.

Schauen wir uns aber zuerst einmal das Speichern eines normalen Strings an. In diesem Fall habe ich auch mit einer Variable vom Typ `string` gearbeitet, um den Namen der Person aus dem Eingabefeld zu lesen:

```
SetLength(username,MAX_PATH);
SetLength(username,GetWindowText(hName,@username[1],MAX_PATH));
```

Dazu muss wohl nichts mehr gesagt werden, denn das Tutorial über Eingabefelder dürfte ja inzwischen bekannt sein. Schauen wir uns daher an, wie der String nun in die INI-Datei gespeichert wird. Wir tragen ihn dabei in die erste Sektion `names` ein. Dieser Sektionsname ist dann auch der erste Parameter der Funktion "`WritePrivateProfileString`":

```
Result := WritePrivateProfileString('name',
```

Da wir in dieser Sektion die Namen doppelt aufführen (und im nächsten Kapitel erkläre ich auch den Grund), benutzen wir nun den zuvor gelesenen Namen zweimal:

```
  @username[1], // Topic (vor dem Gleichheitszeichen)
  @username[1], // Wert (nach dem Gleichheitszeichen)
```

Zum Schluss wird wieder die Datei angeben:

```
  pchar(ExtractFilePath(paramstr(0)) + szIniFile));
```

Es mag verwirrend aussehen, dass in der Anweisung zweimal der Name der Person benutzt wird. Würden wir z.B. eine Nummerierung verwenden, könnte die selbe Anweisung so aussehen:

```
WritePrivateProfileString('name','3',@username[1],'userdata.ini');
```

4.5.3.1. Numerische Werte speichern

Nehmen wir einmal an, wir haben eine Integer-Variable, "`i`", deren Wert wir in der INI-Datei speichern wollen. Da uns dafür keine eigene Funktion zur Verfügung steht, müssen wir den Wert in eine String-Variable übertragen.

Etwas umständlich, aber machbar, ist die Variante, die Integer-Variable mit Hilfe von "`wvsprintf`" in einen Textpuffer zu übertragen und diesen Puffer dann in die INI-Datei zu schreiben. Normalerweise ist der Textpuffer ein `pchar`-Typ (oder ein `array[x..y]of char`), in dem Fall reicht aber wieder ein normaler String:

```
SetLength(s,MAX_PATH);
ZeroMemory(@s[1],MAX_PATH);

wvsprintf(@s[1],'%d',pchar(@i));
```

Leichter geht es mit der Funktion "`inttostr`", mit der Sie den Integer-Wert direkt in einen String konvertieren und in die INI-Datei schreiben lassen:

```
WritePrivateProfileString('test','integertest',pchar(inttostr(i)),'int.ini');
```

Allerdings wird Ihnen diese Funktion nur von der Unit "`SysUtils.pas`" zur Verfügung gestellt. Da wir diese Unit aber weitgehend meiden (weil sie das ausführbare Programm in vielen Fällen nur unnötig vergrößert), bleibt nur die Rückbesinnung auf die "gute alte Zeit":

```
function IntToStr(const i: integer): string;
begin
  Str(i,Result);
end;
```

:o)

4.5.4. Alle Namen einer Sektion lesen

Angesprochen habe ich es bereits: es gibt einen Grund, warum die INI-Datei so aufgebaut ist:

```
[names]
Paul=Paul
Johannes=Johannes
...
```

Natürlich wäre es einfacher, wenn man eine Nummerierung hätte, die man der Reihe nach durchgehen könnte:

```
[names]
1=Paul
2=Johannes
```

Hier könnte man z.B. eine Art Endlosschleife benutzen, die erst verlassen wird, wenn kein zum Schleifenwert passender Topic gefunden wurde.

```
i := 1;
while(true) do begin
  if(GetPrivateProfileString('names',pchar(inttostr(i)),
    nil,buf,BUFSIZE,'userdata.ini') = 0) or
  (buf = nil) then break;

  { ... }

  inc(i);
end;
```

Dies birgt allerdings die Gefahr, dass bei Manipulationen der INI-Datei die Schleife vorzeitig oder evtl. auch nie verlassen wird. Besser ist es daher, wenn man sich nicht darum kümmern muss, wie die Topics der Sektion heißen. Von der Unit "IniFiles.pas" kennen Sie das als Funktion "ReadSection", mit der Sie alle Topicnamen einer Sektion in eine Stringliste eintragen können.

Ohne diese Unit lässt sich die gleiche Funktionalität natürlich auch erreichen. Wir benötigen hierfür einen Puffer. Das Beispiel verwendet dazu eine Größe von 65k. Das sollte eigentlich ausreichen; im Zweifelsfall können Sie ihn auch einen höheren Wert benutzen.

```
const
  BUFSIZE = 65535;
```

Denken Sie daran, dass wir nur die Namen der Topics auslesen wollen; also jeweils die Strings vor dem Gleichheitszeichen:

```
Paul=Paul
```

Des Weiteren brauchen wir eine Art Stringliste, die alle Sektionsnamen aufnehmen soll. Ich denke, ein dynamisches String-Array ist eine gute Wahl.

Zuerst reservieren wir also den Platz für unseren Puffer und setzen das String-Array auf Null:

```
SetLength(kl,0);
GetMem(buf,BUFSIZE);
try
```

Beim Aufruf von "GetPrivateProfileString" ist es nun wichtig, dass wir keinen Topicnamen angeben:

```
  if(GetPrivateProfileString(szNameKey,nil,nil,buf,BUFSIZE,
    pchar(ExtractFilePath(paramstr(0)) + szIniFile)) <> 0) then
  begin
```

Damit befinden sich die Namen aller Topics in unserem Puffer. Sie sind durch das Zeichen #0 voneinander getrennt. Der letzte Name endet mit zwei #0-Zeichen.
Mit Hilfe einer zweiten pchar-Variablen lesen wir nun die einzelnen Namen aus und tragen Sie in unser String-Array ein, das vorher immer entsprechend vergrößert wird:

```
    p := buf;
    while(p^ <> #0) do begin
      SetLength(kl,length(kl)+1);
      kl[length(kl)-1] := p;

      inc(p,lstrlen(p)+1);
    end;
  end;
```

Wie funktioniert es? Die Variable "p" wird auf den Beginn des ermittelten Puffers gesetzt und liefert natürlich nur den ersten String zurück, da das erste #0-Trennzeichen als dessen Ende interpretiert wird. Durch den Aufruf von

```
inc(p,lstrlen(p)+1);
```

wird die Länge des ersten Strings benutzt, um die `pchar`-Variable innerhalb des Puffers zu verschieben. Durch das "`+1`" wird auch das Trennzeichen übersprungen, so dass sich die Variable nun auf dem ersten Zeichen des zweiten Strings befindet. Und so geht es dann weiter bis alle Strings ausgelesen sind.
Danach kann der Puffer dann auch wieder freigegeben werden

```
finally
  FreeMem(buf,BUFSIZE);
end;
```

denn nun steht uns ja unsere Stringliste zur Verfügung:

```
for i := 0 to length(kl) - 1 do
  MessageBox(0,pchar(kl[i]),pchar(inttostr(i+1)),0);
```

Die Topicnamen in dieser Liste benutzen wir nun zum Auslesen der Strings, die uns eigentlich interessieren. Benötigen wir auch die Stringliste nicht mehr, geben wir ihren Speicher ebenfalls wieder frei:

```
SetLength(kl,0);
```

4.5.4.1. Alle Namen und Werte einer Sektion lesen

Wenn wir schon dabei sind, dann wollen wir uns auch noch ansehen, wie wir sowohl Topicnamen als auch den dazu gehörenden Wert auf einmal lesen können. Dazu benutzen wir die Funktion "`GetPrivateProfileSection`".
Problematisch dabei ist, dass der Puffer unter Windows 95/98/ME nur max. 32k (32.767) groß sein darf, während es unter den NT-Systemen keine Beschränkung gibt - obwohl unser Beispiel dort auch "nur" einen 65k großen Puffer verwendet.
Gehen wir also davon aus, wir haben unseren Puffer entsprechend reserviert und initialisiert, dann können wir nun die Funktion aufrufen:

```
GetPrivateProfileSection(szNameKey,
  buf,
  dwLen,
  pchar(ExtractFilePath(paramstr(0)) + szIniFile))
```

Wenn der Puffer nicht ausreicht, um alle Werte aufzunehmen, dann ist das Rückgabeergebnis der Funktion identisch mit der Puffergröße minus Zwei. Sie sollten also darauf achten! Andernfalls enthält der Puffer die Namen und Werte, die durch das Zeichen `#0` voneinander getrennt sind.
Das Auslesen der einzelnen Strings kann wie oben beschrieben durchgeführt werden. Sie benötigen nur eine zusätzliche `pchar`-Variable, um von einem null-terminierten Strings zum nächsten zu gelangen. Angezeigt wird Ihnen dann sowohl der Topicname als auch der Wert, also z.B.

```
Paul=Paul
```

4.5.4.2. Alle Sektionen lesen

Wenn Sie alle Topics und Werte einer Sektion ermitteln können, dann können Sie natürlich auch die Namen aller Sektionen in einer INI-Datei herausfinden. Das entspricht der Funktion "`ReadSections`", die Sie vielleicht von der Unit "`IniFiles.pas`" kennen.

Wie gehabt: wir brauchen auch hier einen Puffer, der die Namen aufnehmen soll. Eine Maximalgröße wird im PSDK nicht genannt. Das Beispiel verwendet den üblichen 65k-Puffer; Sie können im Zweifelsfall einen größeren benutzen.
Mit Hilfe der Funktion "`GetPrivateProfileSectionNames`" lassen sich dann die Namen aller Sektionen in den Puffer schreiben:

```
GetPrivateProfileSectionNames(buf,BUFSIZE,
  pchar(ExtractFilePath(paramstr(0)) + szIniFile));
```

Auch hier sind die einzelnen Namen wieder durch `#0` voneinander getrennt, so dass Sie eine zusätzliche `pchar`-Variable benutzen können, um sie der Reihe nach in eine Stringliste o.ä. einzulesen:

```
p := buf;
while(p^ <> #0) do begin
  SetLength(kl,length(kl)+1);
  kl[length(kl)-1] := p;

  inc(p,lstrlen(p)+1);
end;
```

Ist der Puffer zu klein, entspricht der Rückgabewert der Funktion wieder der Puffergröße minus Zwei.

4.5.5. Werte und Sektionen löschen

Dieses Tutorial wäre nicht komplett, wenn wir nicht auch auf das Löschen von Werten und ganzen Sektionen eingehen würden. Etwas völlig Neues ist hier allerdings auch nicht zu erfahren, denn zum Löschen wird "`WritePrivateProfileString`" benutzt.

Wollen wir den Topic in einer Sektion löschen, geben wir dessen Namen an, lassen aber den Puffer, den wir hier verwendet haben, ungenutzt:

```
WritePrivateProfileString('names','Paul',nil,'userdata.ini');
```

Wollen wir die ganze Sektion "ausradieren", dann lassen wir auch den Namen des Topics leer:

```
WritePrivateProfileString('names',nil,nil,'userdata.ini');
```

4.6. Die Registry

4.6.1. Vorwort

Die Registry ist eine Art Datenbank, in der Anwendungen und Systemkomponenten Konfigurationsdaten speichern, bzw. aus der sie diese abrufen. Zum Zugriff auf die Registry steht ein eigener API-Befehlssatz zur Verfügung. Im Rahmen dieses Beitrags wollen wir auf die gebräuchlichen Funktionen eingehen, so dass wir am Ende ein kleines Programm zur Anzeige der installierten Software besitzen.

Warum gerade so ein Programm?

Zum einen habe ich den Quellcode dazu bereits fertig ... :o). Zum anderen ist es eine schöne Demonstration kleiner Geheimnisse, denn nicht jede installierte Komponente erscheint auch in der Systemsteuerung.

Wir werden das Programm nun aber nicht Zeile für Zeile durchgehen, sondern wir werden uns nur auf die für uns interessanten Teile der Registry konzentrieren. Alles andere (Fenster erzeugen, Toolbar erzeugen, Listview füllen, usw.) setze ich als bekannt voraus, bzw. im Zweifelsfall lässt sich in den entsprechenden Beiträgen die eine oder andere Sache nachschlagen.

Mein Dank geht an Markus Fuchs, Michael Puff und Christian Seehase, für das Lesen und Korrigieren der alten Versionen und die hilfreichen Anregungen und Empfehlungen zur Verbesserung des Programms.

4.6.2. Die Registry öffnen

Die installierte Software befindet sich, wie Sie sicher wissen, im Schlüssel

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall
```

Zum Öffnen dieses Schlüssel benutzen wir die Funktion "`RegOpenKeyEx`". Die Funktion erwartet zunächst einen Root-Schlüssel. Das kann ein zuvor geöffneter Registryschlüssel sein, oder einer der fünf vordefinierten und bekannten Schlüssel. Auf einen solchen müssen wir in dem Fall natürlich zurückgreifen, da wir bisher noch nichts geöffnet haben

```
if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,
```

Der nächste Parameter ist der gewünschte Pfad. Das Beispiel benutzt hier zwar eine Konstante, aber zur besseren Übersicht soll hier der tatsächliche Pfad genannt werden

```
  'Software\Microsoft\Windows\CurrentVersion\Uninstall',
```

Der nächste Parameter ist reserviert und daher Null

```
  0,
```

Nun geben wir an, wie wir auf den Schlüssel zugreifen wollen. Ausgehend von der Tatsache, dass wir die Namen der installierten Software nur lesen aber nicht ändern wollen, genügt uns hier die Option

```
 KEY_READ,
```

`KEY_READ` ist ohnehin aus Sicherheitsgründen vorzuziehen. Da mit Windows XP allmählich ein System auf NT-Basis im Heimbereich Verbreitung findet, darf die Rechte-Frage meiner Meinung nach nicht außer Acht gelassen werden. Bei 9x-Systemen kann im Prinzip jeder Benutzer alles machen; Gutes wie Schlechtes. Bei NT-Betriebssystemen entscheiden die Rechte darüber, was ein Benutzer darf und was nicht. Im Normalfall hat auch nur ein Administrator vollen Zugriff auf den Schlüssel `HKEY_LOCAL_MACHINE`. Nun ist aber nicht auszuschließen, dass ein "normaler" Benutzer das Programm startet. Wenn Ihr Programm z.B. davon abhängig ist, Daten aus diesem Schlüssel zu lesen und in diesen zu speichern, dann sollten Sie es beenden, wenn keine ausreichenden Rechte vorhanden sind.
In unserem Fall kann diese Prüfung vernachlässigt werden, da wir hier die Daten ja nur lesen wollen. Und dafür genügt die o.g. Option.

Der letzte Parameter ist eine HKEY-Variable, die das Handle des geöffneten Schlüssels enthält. Diese Variable nutzen wir für die folgenden Lese- und (in anderen Fällen!) Schreibzugriffe.

```
  rgHandle) = ERROR_SUCCESS) then
try
  { ... }
finally
  RegCloseKey(rgHandle);
end;
```

Neben der Prüfung, ob der Zugriff (in dem Fall: das Öffnen des Schlüssels) erfolgreich war, empfiehlt es sich, die nun folgenden Anweisungen in einen **try**-**finally**-Block einzuklammern. Im Fehlerfall wird der Schlüssel dann wenigstens noch mit

```
RegCloseKey(rgHandle);
```

geschlossen.

RegOpenKeyEx-Definition

```
LONG RegOpenKeyEx(
  HKEY hKey,          // Handle eines zuvor geöffneten Schlüssels, bzw.
                      // HKEY_CLASSES_ROOT, usw.
  LPCTSTR lpSubKey,   // Pfad
  DWORD ulOptions,    // reserviert (Null)
  REGSAM samDesired, // Zugriffsrechte
  PHKEY phkResult     // HKEY-Variable mit dem Schlüssel-Handle für weitere
Zugriffe
);
```

RegCloseKey-Definition

```
LONG RegCloseKey(
  HKEY hKey           // geöffneter Schlüssel
);
```

Die wichtigsten Zugriffsrechte

Wert	Bedeutung
KEY_READ	kombiniert die Standardrechte (DELETE, READ_CONTROL, WRITE_DAC, WRITE_OWNER, SYNCHRONIZE) mit KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS und KEY_NOTIFY
KEY_QUERY_VALUE	erforderlich um Werte auslesen zu können
KEY_ENUMERATE_SUB_KEYS	erforderlich um vorhandene Schlüssel "aufzählen" zu können
KEY_WRITE	kombiniert die Standardrechte mit KEY_SET_VALUE und KEY_CREATE_SUB_KEY
KEY_SET_VALUE	erforderlich um einen Registrywert erzeugen, löschen oder ändern zu können
KEY_CREATE_SUB_KEY	erforderlich um einen Schlüssel anlegen zu können
KEY_ALL_ACCESS	kombiniert die Standardrechte mit den Rechten von KEY_READ und KEY_WRITE sowie mit KEY_CREATE_LINK

Hinweis

Wenn Sie die "Registry"-Unit verwenden, wird die Registry durch einen Aufruf von

```
reg := TRegistry.Create;
{ ... }
```

standardmäßig mit KEY_ALL_ACCESS geöffnet. Das kann, insbesondere auf NT-Systemen, zu den bereits geschilderten Rechte-Problemen führen, wenn Sie auf Schlüssel zugreifen, die nicht für jeden Benutzer zugänglich sind. Aber auch hier lässt sich ein anderes Zugriffsrecht verwenden. Es wird auch in der Delphi-Hilfe erwähnt, nur in den meisten Fällen leider überlesen oder sogar ignoriert. Unser API-Beispiel vom Anfang könnte z.B. so aussehen:

```
reg := TRegistry.Create(KEY_READ);
with reg do
  try
    RootKey := HKEY_LOCAL_MACHINE;
    if(OpenKey('Software\Microsoft\Windows\CurrentVersion\Uninstall',
      false)) then
    try
      { ... }
    finally
      CloseKey;
    end;
  finally
    reg.Free;
  end;
```

4.6.3. Die vorhandenen (Unter-)Schlüssel enumerieren

Im letzten Kapitel haben wir die Registry geöffnet und befinden uns nun im Hauptpfad der installierten Software. Da wir nicht wissen können, welche Software der Anwender installiert hat, brauchen wir eine Möglichkeit, die vorhandenen Schlüssel der Reihe nach auszulesen.

Diese Möglichkeit heißt "RegEnumKeyEx".

Zur Ausführung benötigen wir zunächst natürlich ein Zeichenarray (pchar). Es bleibt dabei Ihnen überlassen, wie Sie vorgehen. Sie können das Array so deklarieren:

```
var
  pBuf : array[0..MAX_PATH]of char;
```

Oder Sie ermitteln die Größe des Puffers zur Laufzeit und deklarieren die Variable dann so:

```
var
  pBuf : pchar;
```

Es funktioniert deshalb beides, weil (laut PSDK) die Länge für einen Schlüsselnamen max. 255 Zeichen beträgt. Und wenn Sie ein statisches Array verwenden wollen (s. erste Deklaration von "pBuf"), umfasst dieses durch MAX_PATH ja sowieso 260 Zeichen.
Aufpassen sollten Sie, wenn Sie mit der Unicode-Version, "RegEnumKeyExW", arbeiten wollen. Sie liefert einen Zeiger auf ein WideChar-Array, bzw. sie erwartet als Puffer ein solches Array; und hier belegt jedes Zeichen im Array zwei Bytes. Sie sollten also die doppelte Puffergröße verwenden.

Neben dem Zeichenarray benötigen wir eine dword-Variable, die die Länge des Puffers angibt. Diese Länge muss vor jedem neuen Aufruf von "RegEnumKeyEx" auf den Maximalwert des Puffers zurückgesetzt werden! Warum?
Die Variable wird an die Funktion übergeben und von dieser auf die tatsächliche Länge des ermittelten Schlüsselnamens gesetzt. Nehmen wir als Beispiel an, der ermittelte Name lautet "IE40". Dann wird der Wert der dword-Variablen auf 4 gesetzt, da ja nur 4 Zeichen kopiert wurden.
Würden wir die Funktion nun ein zweites Mal aufrufen und dabei die Länge nicht auf die Puffergröße zurücksetzen, bekäme "RegEnumKeyEx" die Anweisung, den nächsten Schlüsselnamen in einen 4 Bytes großen Puffer zu kopieren.

Wenn der nächste Name allerdings mehr Zeichen aufweist, wäre der Fehler ERROR_MORE_DATA die Folge.

Und zu guter Letzt benötigen wir eine Zählervariable, die wir auf Null setzen und solange um Eins erhöhen, bis die Funktion ERROR_NO_MORE_ITEMS als Ergebnis zurückliefert.

Schauen wir uns zunächst das einfachere Beispiel an, bei dem wir ein "array[0..MAX_PATH]of char" als Puffer verwenden:

```
i := 0;

while(true) do begin
  ZeroMemory(@pBuf,sizeof(pBuf));
  dwLen   := sizeof(pBuf);
```

Wie bereits angesprochen: wir setzen die Zählervariable "i" auf Null, da sie als Ausgangspunkt unserer Enumeration dient. Dann treten wir in eine Endlosschleife ein, leeren den Puffer und setzen die dword-Variable ("dwLen") auf die Puffergröße.

Jetzt rufen wir die Funktion auf und geben zuerst das Handle unseres geöffneten Schlüssels an:

```
  retcode := RegEnumKeyEx(rgHandle,
```

Es folgt unsere Zählervariable

```
    i,
```

unser Puffer

```
    pBuf,
```

und dessen Größe

```
    dwLen,
```

Die vier weiteren Parameter sind für uns nicht von Bedeutung und werden von uns daher auch ignoriert:

```
    nil, nil, nil, nil);
```

Wenn es keine weiteren Schlüssel gibt, dann sollten wir die Schleife natürlich verlassen:

```
  if(retcode = ERROR_NO_MORE_ITEMS) then break;
```

Wenn die Funktion erfolgreich ausgeführt werden konnte, dann befindet sich nun der Name des ersten bzw. nächsten Unterschlüssels im Zeichenpuffer, und wir können ihn in unsere Liste eintragen oder anderweitig be- oder verarbeiten

```
  if(retcode = ERROR_SUCCESS) then begin
    { ... }
  end;
```

Und damit wir kein "hängendes" Programm basteln, das sich auf Nimmerwiedersehen verabschiedet, sollten wir unbedingt unsere Zählervariable erhöhen, damit wir irgendwann den Fehler ERROR_NO_MORE_ITEMS provozieren und die Schleife verlassen können.

```
  inc(i);
end;
```

Sie können auch auf Nummer Sicher gehen und nur auf ERROR_SUCCESS prüfen; sollte diese Prüfung negativ verlaufen, verlassen Sie die Schleife mit "break". Auf die Weise ist sichergestellt, dass andere evtl. auftretende und unvorhergesehene Fehler Ihr Programm nicht abschießen.

RegEnumKeyEx-Definition

```
LONG RegEnumKeyEx(
  HKEY hKey,                    // HKEY-Variable mit Handle zum
                                // geöffneten Schlüssel
  DWORD dwIndex,                // null-basierender Zähler
  LPTSTR lpName,                // Zeiger auf PChar-Puffer
  LPDWORD lpcName,              // Puffergröße
  LPDWORD lpReserved,           // reserviert (nil)
  LPTSTR lpClass,               // Puffer für die Klassennamen
  LPDWORD lpcClass,             // Größe des Klassennamenpuffers
  PFILETIME lpftLastWriteTime   // Zeit der letzten Schreibaktion
);
```

4.6.4. RegQueryInfoKey, ... enumerieren zum Zweiten ...

Das vorige Kapitel zeigte, wie man die Schlüsselnamen mit einem festen Puffer auslesen kann. Als Ergänzung wollen wir uns nun kurz die Funktion "RegQueryInfoKey" ansehen, die wir bei der dynamischen Variante benötigen.
Das Beispielprogramm verwendet diese Funktion aber auch, um die Anzahl der vorhandenen Schlüssel für die Fortschrittsanzeige zu ermitteln. Immerhin soll der Anwender ja merken, dass noch etwas passiert.

RegQueryInfoKey-Definition

```
LONG RegQueryInfoKey(
  HKEY hKey,                      // Handle des geöffneten Schlüssels
  LPTSTR lpClass,                 // Puffer für Klassennamen
  LPDWORD lpcClass,               // Größe des Klassennamen-Puffers
  LPDWORD lpReserved,             // reserviert (nil)
  LPDWORD lpcSubKeys,             // Anzahl der Unterschlüssel
  LPDWORD lpcMaxSubKeyLen,        // Anzahl der Zeichen im längsten Namen
  LPDWORD lpcMaxClassLen,         // Anzahl der Zeichen des längsten
                                  // Klassennamens
  LPDWORD lpcValues,              // Anzahl der Werte im Schlüssel
  LPDWORD lpcMaxValueNameLen,     // Anzahl der Zeichen im längsten
                                  // Werte-Namen
  LPDWORD lpcMaxValueLen,         // Größe des größten Wertes
  LPDWORD lpcbSecurityDescriptor, // security descriptor
  PFILETIME lpftLastWriteTime     // Zeitpunkt der letzten Schreibaktion
);
```

Weil uns nicht alle Parameter dieser Funktion interessieren, benötigen wir hier lediglich zwei dword-Variablen für die Anzahl der Schlüssel und die Anzahl der Zeichen im längsten Schlüsselnamen:

```
if(RegQueryInfoKey(rgHandle,nil,nil,nil,@count,@pBufLen,nil,
  nil,nil,nil,nil,nil) <> ERROR_SUCCESS) or
  (count = 0) then goto Stop;
```

In diesem Fall wäre "pBufLen" die Variable, die die Zeichenanzahl des längsten Schlüsselnamens aufnimmt. Diesen Wert nutzen wir nun und reservieren entsprechend Speicher:

```
GetMem(pBuf,pBufLen+1);
try
  { ... }
finally
  FreeMem(pBuf,pBufLen+1);
end;
```

Der Rest ähnelt dann wieder der Vorgehensweise beim festen Puffer. Ich spare mir daher die nochmalige Beschreibung. Außerdem demonstriert Ihnen das Beispielprogramm die dynamische Variante.

Werfen wir daher noch kurz einen Blick auf die Variable "count", die im Beispiel die Anzahl der Schlüssel enthält. Dieser Wert wird für die Fortschrittsanzeige benötigt. Dazu erzeugt das Programm schlicht einen kleinen Dialog, der nur einen Progressbar enthält. An den senden wir nur die Nachricht "PBM_SETPOS" und übermitteln dabei die aktuelle Position (in der Variablen "i") und die maximale Anzahl:

```
SendDlgItemMessage(dlg,130,PBM_SETPOS,
  MulDiv(i,100,count),0);
```

Und was ist mit den Werten in einem Schlüssel?

Mit dieser Frage beschäftigen wir uns im Kapitel Schlüssel kopieren bzw. verschieben zu einem späteren Zeitpunkt.

4.6.5. Werte aus der Registry lesen

Machen wir einen Schritt zurück und rufen uns den Teil in Erinnerung, in dem wir (hoffentlich erfolgreich!) die Funktion "RegEnumKeyEx" zum Ermitteln der Schlüsselnamen verwendet haben. Immer wenn der Funktionsaufruf erfolgreich war, öffnet das Beispielprogramm den ermittelten Schlüssel direkt, was verkürzt so aussieht:

```
if(retcode = ERROR_SUCCESS) and
  (RegOpenKeyEx(rgHandle,pBuf,0,KEY_READ,ukey) = ERROR_SUCCESS) then
try
  { ... }
finally
  RegCloseKey(ukey);
end else break;
```

Die nachfolgenden Erläuterungen spielen sich zwischen dem **try** und **finally** ab, und daher ist auch "ukey" das Handle, das wir ab jetzt benutzen müssen. Das Beispielprogramm benutzt eine eigene Funktion zum Auslesen der Werte. Auf diese Weise müssen die selben Anweisungen nicht ständig neu geschrieben werden; zumal die Funktion auch gleichzeitig eine Art "value exists"-Prüfung enthält.

Und das ist auch gleich das erste, das wir uns ansehen wollen. Unser jeweiliger Unterschlüssel ist ja nun geöffnet, und nun wollen wir den Displaynamen (den Sie in der Systemsteuerung sehen) und den Uninstall-String auslesen. Zuerst sollten wir aber sicherstellen, dass der Wert überhaupt existiert, dass er dem erwarteten Typ entspricht, und dass er auch Inhalt hat. Dazu benötigen wir zwei dword-Variablen

```
var
  lpType,
  cbData : dword;
```

Zum Zugriff auf Werte steht uns die API-Funktion "RegQueryValueEx" zur Verfügung, die wir auch gleich zur besagten Prüfung verwenden können. Die Funktion erwartet neben dem obligatorischen Schlüssel-Handle und dem Namen des gesuchten Wertes auch Angaben zu dessen Typ.
Zuerst ein wenig Vorarbeit:

```
lpType := REG_NONE;
cbData := 0;
```

Mit diesen beiden Zeilen setzen wir zuerst einen nicht definierten (allgemeinen) Typ, damit wir auch wirklich jeden Eintrag im Schlüssel berücksichtigen der den gesuchten Namen trägt. Die Puffergröße wird auf Null gesetzt, den Puffer selbst ignorieren wir in diesem Fall erst einmal. Dann rufen wir die Funktion auf und übergeben zuerst wieder das Handle auf den Schlüssel

```
RegQueryValueEx(ukey
```

An zweiter Stelle kommt der Name des Wertes, der uns interessiert

```
  'DisplayName',
```

Der dritte Parameter wird nicht benutzt

```
  nil,
```

Nun folgt unser vorher festgelegter, allgemeiner Typ

```
  @lpType,
```

Den Puffer ignorieren wir (wie schon gesagt)

```
  nil,
```

Und zuletzt geben wir die Variable an, die die Puffergröße enthält.

```
  @cbData)
```

Was passiert nun? - Im Idealfall gibt es einen Wert mit dem gesuchten Namen. Unserer Variablen "`lpType`" wird nun der tatsächliche Typ des gefundenen Wertes zugewiesen.

Wert	Bedeutung
REG_BINARY	binäre Form der Daten
REG_DWORD	32-Bit-Wert
REG_SZ	null-terminierter String
REG_EXPAND_SZ	null-terminierter String, der Referenzen auf Umgebungsvariablen enthalten kann (z.B. "`%PATH%`")

(im PSDK finden Sie weitere Typen)

Die Variable "`cbData`" enthält nach dem Funktionsaufruf die Anzahl der kopierten oder benötigten Bytes. Wenn Sie einen Wert angeben, der kleiner als der benötigte Puffer ist, dann gibt die Funktion den Fehler `ERROR_MORE_DATA` zurück, ansonsten ist das Ergebnis `ERROR_SUCCESS`. Damit wäre der weitere Weg bereits festgelegt:

```
if(RegQueryValueEx(ukey,'DisplayName',nil,@lpType,nil,@cbData) = ERROR_SUCCESS)
and
  (lpType in[REG_SZ,REG_EXPAND_SZ]) and
  (cbData > 0) then
begin
  { ... }
end;
```

Im Fall des Beispielprogramms halte ich es für zweckmäßig sowohl `REG_SZ` als auch `REG_EXPAND_SZ` als Typ zu berücksichtigen. Es wäre ja möglich, dass ein Uninstall-String in letzterem Format gespeichert wird bzw. werden muss, weil er eine Referenz auf eine Umgebungsvariable enthält.

Wenn also unsere Prüfung geklappt hat, dann setzen wir den Puffer auf die gewünschte Länge und rufen die Funktion erneut auf:

```
SetLength(s,cbData);
if(RegQueryValueEx(ukey,'DisplayName',nil,nil,
  @s[1],@cbData) = ERROR_SUCCESS) then SetLength(s,cbData-1)
else s := '';
```

Unser Puffer ist hier eine `string`-Variable, die wir diesmal beim Funktionsaufruf natürlich auch angeben. Im Erfolgsfall enthält "`s`" nun den Inhalt des ausgelesenen Wertes.

RegQueryValueEx-Definition

```
LONG RegQueryValueEx(
  HKEY hKey,           // Handle des geöffneten Schlüssels
  LPCTSTR lpValueName, // Name des gesuchten Wertes
  LPDWORD lpReserved,  // reserviert (nil)
  LPDWORD lpType,      // Typ des Wertes
  LPBYTE lpData,       // Zeiger auf Puffer
  LPDWORD lpcbData     // Puffergröße
);
```

Nicht unerwähnt bleiben darf und soll, dass dieser Weg nicht unter Windows NT, 2000 und XP funktioniert, wenn Sie auf Daten im Schlüssel `HKEY_PERFORMANCE_DATA` zugreifen. Hier würde die Funktion den Fehler `ERROR_MORE_DATA` auslösen, aber nicht die benötigte Puffergröße in die `dword`-Variable schreiben.
Das liegt daran, dass sich die Daten im genannten Schlüssel von einem Aufruf zum nächsten ändern können. Microsoft schlägt daher im PSDK eine Art Schleife vor, in der Sie den Wert der `dword`-Variablen solange erhöhen, bis die Funktion erfolgreich ist. Beispielsweise

```
cbData := 0;

while(RegQueryValueEx(ukey,'DynData',nil,@lpType,nil,@cbData) <>
  ERROR_SUCCESS) do inc(cbData);
```

Die zweite Möglichkeit wäre, die erforderlichen Puffergrößen über "`RegQueryInfoKey`" zu ermitteln. Etwa:

```
RegQueryInfoKey(rgHandle,nil,nil,nil,nil,nil,nil,
  nil,nil,@cbData,nil,nil);
```

In unserem Fall können wir dieses Problem zumindest vernachlässigen, weil wir zu keinem Zeitpunkt auf den Schlüssel `HKEY_PERFORMANCE_DATA` zugreifen.

4.6.5.1. Und wenn es keine Strings sind?

Für unser Beispielprogramm benötigen wir nun zwar nur Strings, aber Sie stehen vielleicht einmal vor der Aufgabe, andere Formate auslesen zu müssen. Als Beispiel möchte ich den Registryeintrag

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\No
DriveTypeAutoRun
```

anführen. Normalerweise ist dieser Wert "`0x00000095`". Daher startet z.B. ein Programm automatisch, sobald Sie eine CD-ROM einlegen (sofern die CD entsprechend ausgerüstet ist, natürlich!). Würden Sie den Wert auf "`0x000000b5`" ändern, wäre der Autostart der CDs unterbunden. Das Interessante an diesem Wert ist, dass er unter Windows 98 als Binärwert gespeichert ist, unter Windows XP jedoch als `dword`.
Das ändert nun aber nichts am Prinzip; wir müssen lediglich das Betriebssystem prüfen und den passenden Typ vergleichen. Zum Auslesen selbst können wir in beiden Fällen eine `dword`-Variable benutzen, da ja auch die binäre Form (Win98) nur 4 Bytes groß ist. Das funktioniert, weil der Binärwert (wie in einer typischen Datei unter Windows) mit dem niederwertigen Byte zuerst gespeichert wird. Hätten wir z.B. eine `dword`-Variable mit dem Hex-Wert "`0x01020304`", würde dies in einer Datei (und auch im binären Format in der Registry) so aussehen:

```
04 03 02 01
```

Dieses "umgedrehte" Format interessiert uns allerdings nicht, weil wir ja nicht direkt auf die Festplatte zugreifen sondern Funktionen des Betriebssystems dafür verwenden. Also ist es hier Sache von Windows, in welcher Form solche Daten gespeichert werden. Müssten wir den Wert nun aus einer Datei lesen, bräuchten wir nur den Offset und könnten dann mit folgender Beispielanweisung aktiv werden:

```
BlockRead(f,AutorunSettings,sizeof(dword));
```

Windows sorgt im Hintergrund dafür, dass die Variable den richtigen Wert (quasi "richtig herum") enthält. Und beim binären Format der Registry ist es auch so, weshalb wir in beiden Fällen (ob nun `REG_DWORD` oder `REG_BINARY`) die

selbe dword-Variable nutzen können.

Wir prüfen also, wie gehabt, ob der Funktionsaufruf erfolgreich war

```
if(RegQueryValueEx(rgHandle,'NoDriveTypeAutoRun',nil,@lpType,nil,@cbData) =
ERROR_SUCCESS) and
```

Dann sollten wir prüfen ob der Typ ein Binärwert ist

```
  (((lpType = REG_BINARY) or
```

oder ein dword-Wert; **wenn** Windows XP läuft

```
   ((lpType = REG_DWORD) and (WinXP))) and
```

Und natürlich sollte unser Wert genau 4 Bytes (die Größe eines dword) umfassen

```
  (cbData = 4) then
```

Dann (und nur dann) sollten wir die Funktion erneut aufrufen und diesmal unsere Variable als Zeiger angeben:

```
RegQueryValueEx(rgHandle,'NoDriveTypeAutoRun',nil,@lpType,@AutorunSettings,@cbDa
ta);
```

4.6.6. Werte in die Registry schreiben

Werfen wir doch mal einen Blick auf das Beispielprogramm.
Was macht es?

Es liest die Displaynamen und die Uninstall-Strings der installierten Programme aus. So weit, so gut. Zusätzlich liest es aber auch die versteckten Komponenten, die nicht in der Systemsteuerung zu sehen sind, und deren Registryeinträge "QuietDisplayName" und "QuietUninstallString" lauten. Das ist das ganze Geheimnis! Deswegen zeigt Ihnen das Programm auch mehr an als Ihre Systemsteuerung.

Diese Fähigkeit des Systems können wir nutzen. Wenn Sie z.B. den Displaynamen einer Software löschen, verschwindet der Eintrag aus der Systemsteuerung, obwohl die Software natürlich noch vorhanden ist und deinstalliert werden kann. Das ist übrigens auch die zweite Aufgabe des Programms. Sie können nicht nur den Displaynamen (beide; den bekannten und den versteckten) nach Belieben ändern und so Ihre Liste vergrößern oder verkleinern, Sie können Programme auch direkt deinstallieren lassen.
Mir ist es schon einmal passiert, dass ich in der Systemsteuerung den falschen Eintrag angeklickt und damit die falsche Software entfernt habe. Seitdem nutze ich das Programm bevorzugt, um die Dinge zu verbergen, die man nicht versehentlich entfernen können soll.

Gut. Nach der Werbung für meine Software wollen wir uns nun wieder auf das Wesentliche konzentrieren. ;o)

4.6.6.1. Den Registryschlüssel mit neuen Zugriffsrechten öffnen

Die Überschrift bringt es bereits auf den Punkt: um Werte in die Registry schreiben zu können, genügt KEY_READ nicht mehr. Erschwerend kommt hier noch dazu, dass wir in den Schlüssel HKEY_LOCAL_MACHINE schreiben müssen.
Ich erwähnte ja bereits die Frage der Rechte bei NT-Betriebssystemen (Windows NT, 2000, XP). Das Beispielprogramm ist so gestaltet, dass es diese Funktionalität nur anbietet, wenn es von einem Benutzer mit Administratorrechten gestartet wurde. Andernfalls sind die Toolbar-Buttons z.B. deaktiviert, usw. Nun könnten Sie zwar den Quellcode entsprechend ändern, aber ich möchte davon abraten. Es bringt nichts; bestenfalls sehen Sie ein paar Fehlermeldungen, aber Abstürze oder Schäden an der Registry wären ebenso denkbar ...

Setzen wir also voraus, dass wir die notwendigen Rechte seitens des Betriebssystems haben. Nun öffnen wir den Registryschlüssel mit dem zusätzlichen Recht KEY_WRITE

```
if(RegOpenKeyEx(HKEY_LOCAL_MACHINE,pchar(szUnInst + '\' + buffer),0,
  KEY_READ or KEY_WRITE,rgHandle) = ERROR_SUCCESS) then
try
  { ... }
finally
  RegCloseKey(rgHandle);
end;
```

Der Pfad ergibt sich in dem Fall aus dem Standardpfad der installierten Software (in der Konstante `szUnInst`) und dem ausgewählten Eintrag der Listview.

Bleiben wir beim Displaynamen -
Im Beispiel wird er aus dem Dialog gelesen, den das Programm bereitstellt. Gehen wir davon aus, dass der Anwender einen neuen Wert eingetragen hat. Dann rufen wir nun die Funktion "`RegSetValueEx`" auf, um den neuen Wert in die Registry zu schreiben. Und auch hier steht an erster Stelle das Handle unseres eben geöffneten Schlüssels:

```
RegSetValueEx(rgHandle,
```

Es folgt der Name, den wir eintragen oder ändern wollen

```
  'DisplayName',
```

Der nächste Parameter ist reserviert und muss daher Null sein

```
  0,
```

Es folgt der Typ unseres Wertes. Da wir einen String eintragen wollen, benutzen wir

```
  REG_SZ,
```

Der nächste Parameter ist ein Zeiger auf den Puffer, der den Inhalt unseres Eintrags enthält. Da wir hier einen String verwenden, verweisen wir auf das erste Zeichen (denn im 0. Zeichen des Strings befindet sich dessen Länge)

```
  @dn[1],
```

Der letzte Parameter ist die Größe des Puffers; die Stringlänge plus 1 in unserem Fall

```
  length(dn)+1)
```

Ich habe es mir hier ein wenig einfach gemacht und nur den Funktionsaufruf demonstriert. Im Programm findet natürlich eine Prüfung statt, ob der Aufruf auch erfolgreich war. Sie sollten dies ebenfalls nicht vernachlässigen.

RegSetValueEx-Definition

```
LONG RegSetValueEx(
  HKEY hKey,              // Handle des geöffneten Schlüssels
  LPCTSTR lpValueName,    // Name des Eintrags
  DWORD Reserved,         // Null
  DWORD dwType,           // Typ des Eintrags
  const BYTE* lpData,     // Zeiger auf den Puffer
  DWORD cbData            // Größe des Puffers
);
```

4.6.6.2. Werte löschen

Um einen Eintrag zu löschen, benötigen wir lediglich unseren geöffneten Schlüssel. Das Beispielprogramm demonstriert Ihnen die Vorgehensweise: wenn die Variable "`dn`" der Konstante `szUnknown` ("[unbekannt]") entspricht, wird sie auf Null zurückgesetzt. Ist der String leer (weil im Dialog nur der Eintrag entfernt und nicht durch einen neuen ersetzt wurde), ist dies natürlich nicht erforderlich. Das Programm prüft dann ob die Variable leer ist und entfernt in diesem Fall den Eintrag mit der Funktion "`RegDeleteValue`":

```
if(dn = '') then iRes := RegDeleteValue(rgHandle,'DisplayName')
{ ... }
```

Die Syntax der Funktion ist also recht einfach.

RegDeleteValue-Definition

```
LONG RegDeleteValue(
  HKEY hKey,            // Handle des geöffneten Schlüssels
  LPCTSTR lpValueName   // Name des Eintrags
);
```

4.6.6.3. Noch einmal die Frage: was, wenn es keine Strings sind?

Kommen wir noch einmal auf den hier angesprochenen Autostart-Eintrag zurück, der entweder binär oder als `dword` gespeichert wird.

Die eben beschriebene Vorgehensweise ändert sich auch hier nicht. Anstelle des Zeigers auf einen String, benutzen wir hier einen Zeiger auf unsere `dword`-Variable. Als Größe übergeben wir den Größenwert eines `dword`, bzw. den Größenwert der von uns benutzten Variable. Und der Typ entscheidet, in welchem Format der Wert gespeichert wird.
Beispiel A: wir speichern den Autostart-Eintrag für Windows 98 im binären Format:

```
RegSetValueEx(rgHandle,
  'NoDriveTypeAutoRun',
  0,
  REG_BINARY,
  @AutorunSettings,
  sizeof(AutorunSettings));
```

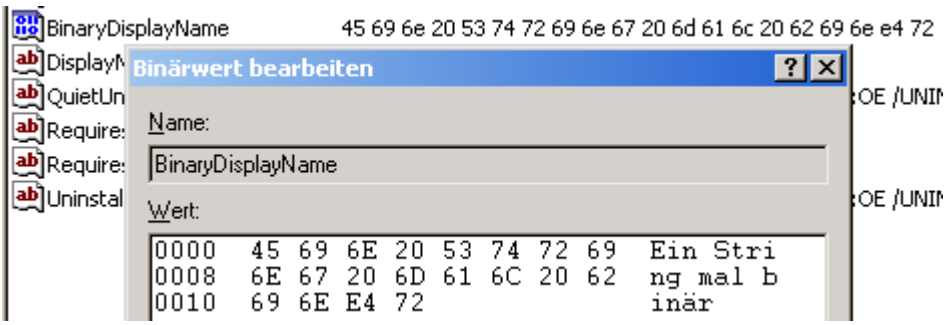
Beispiel B: wir speichern den Eintrag nun als `dword`. Es ist alles identisch, nur der Typ ist ein anderer:

```
RegSetValueEx(rgHandle,
  'NoDriveTypeAutoRun',
  0,
  REG_DWORD,
  @AutorunSettings,
  sizeof(AutorunSettings));
```

Was würde passieren, wenn wir unseren String im binären Format speichern? Um Ärger zu vermeiden verwenden wir aber einen anderen Namen:

```
RegSetValueEx(rgHandle,'BinaryDisplayName',0,
  REG_BINARY,@dn[1],length(dn)+1);
```

Wie gesagt: "`dn`" ist eigentlich eine String-Variable, aber das Ergebnis sieht im Registry-Editor so aus:



Es klappt also. Dennoch möchte ich davon abraten, wild mit den Formaten zu jonglieren. In dem Fall mag es

funktionieren, weil man einen String ja durchaus in dieser binären Form ablegen kann; in anderen Fällen kann es Fehler geben, mehr oder weniger schwerwiegend ... aber stets unschön ...

4.6.7. Schlüssel löschen

Um einen Registryschlüssel zu löschen benötigen wir nur die Funktion "RegDeleteKey". Hierbei hat die Sorgfalt absoluten Vorrang vor allem anderen. Wenn Sie einen falschen String angeben, etwas vergessen oder zuviel anhängen, besteht die Gefahr, dass Sie die Registry schädigen und so Ihr Betriebssystem in die Knie zwingen ... evtl. endgültig ...

Das Beispielprogramm fragt deshalb vor dem Löschen eines Schlüssels nach und zeigt auch den zu löschenden Schlüssel an. Ich gehe davon aus, dass Sie wissen was Sie tun. Als Anwender sollten Sie vor dem Löschen genau lesen und nachdenken ob es wirklich der Schlüssel ist, den Sie entfernen wollen. Als Entwickler stehen Sie meiner Meinung nach an diesem Punkt in der Pflicht, so viele Informationen wie möglich zu liefern. Anstelle des bloßen Schlüsselnamens sollten Sie daher den kompletten Pfad angeben. Und Sie sollten auch die Abfrage von Anfang an auf Nein einstellen, für den Fall, dass ein User versehentlich auf Enter drückt. Im Programm sieht das so aus:

```
if(MessageBox(0,
  pchar('Wollen Sie den Schlüssel "' + szUnInst + '\' + buffer + '" wirklich
entfernen?'),
  'UnInstall Secrets',
  MB_YESNO or MB_ICONQUESTION or MB_DEFBUTTON2) = ID_YES) then
begin
end;
```

Das Programm präsentiert Ihnen den kompletten Pfad und stellt außerdem mit MB_DEFBUTTON2 den zweiten Button der Dialogbox als aktiv ein; in dem Fall wäre das der "Nein"-Button, und damit sind Sie auf der sicheren Seite.

Haben Sie sich entschieden, den Schlüssel zu löschen, rufen Sie die o.g. Funktion auf und geben zuerst das Handle eines evtl. übergeordneten Schlüssels an (in unserem Fall HKEY_LOCAL_MACHINE als Root) und natürlich den Namen des zu löschenden Pfades:

```
if(RegDeleteKey(HKEY_LOCAL_MACHINE,
  pchar(szUnInst + '\' + buffer)) = ERROR_SUCCESS) then
ListView_DeleteItem(hListview,i);
```

Im Erfolgsfall wird der Schlüssel dann auch gleich aus der Listview entfernt.

RegDeleteKey-Definition

```
LONG RegDeleteKey(
  HKEY hKey,         // Handle des geöffneten Schlüssels
  LPCTSTR lpSubKey   // Name des Schlüssels, der gelöscht werden soll
);
```

Unter NT, 2000 und XP darf der zu löschende Schlüssel allerdings keine Unterschlüssel haben! Im Fall unseres Beispiels sollte das keine Rolle spielen, da Einträge von installierter Software normalerweise keine Unterschlüssel haben ... Aber: sicher ist sicher. Also ändern wir den o.g. Code um und benutzen eine neue Prozedur. Die öffnet den jeweiligen Schlüssel, prüft ihn auf die Existenz möglicher Unterschlüssel und ruft sich in dem Fall selbst wieder auf:

```
// Schlüssel öffnen
if(RegOpenKeyEx(parent,@szKeyName[1],0,KEY_READ,reg) = ERROR_SUCCESS) then
try
  // Anzahl der Unterschlüssel herausfinden
  if(RegQueryInfoKey(reg,nil,nil,nil,@dwSubKeys,nil,
    nil,nil,nil,nil,nil,nil) = ERROR_SUCCESS) and
  // es sind mehr als Null
    (dwSubKeys > 0) then
  for i := 0 to dwSubKeys - 1 do begin
    ZeroMemory(@buf,sizeof(buf));
    dwLen   := MAX_PATH;

    // jeden Schlüsselnamen auslesen
    if(RegEnumKeyEx(reg,i,buf,dwLen,nil,nil,nil,nil) = ERROR_SUCCESS) and
      (dwLen > 0) then
    // und die Funktion neu aufrufen
    DelRecurse(reg,buf);
  end;
finally
  RegCloseKey(reg);
end;
```

Sind keine weiteren Schlüssel vorhanden, wird der jeweils aktuelle geschlossen (natürlich!) und dann mit

```
Result := RegDeleteKey(parent,@szKeyName[1]);
```

gelöscht.

4.6.8. Schlüssel kopieren bzw. verschieben

Wozu soll man das überhaupt können?
Im Beispielprogramm haben Sie die Möglichkeit, den Namen eines Schlüssels zu ändern. Dazu klicken Sie einen Eintrag in der Listview an und drücken F2, bzw. Sie klicken noch einmal, um den "in place"-Editor der Listview zu aktivieren. Wenn Sie den Namen dann ändern, muss dieser natürlich irgendwie in die Registry kommen.

Wäre es ein normaler Wert (= Eintrag), kein Problem: Sie hätten den alten Wert gelöscht und mit dem neuen Namen wieder eingetragen. Hier handelt es sich aber um Schlüssel, die wiederum Werte und Unterschlüssel enthalten können. Zugegeben, Schlüssel sind in diesem speziellen Fall sicher nicht zu erwarten, aber wenn Sie einen anderen Registryschlüssel kopieren oder verschieben wollen, dann sollten Sie damit rechnen.

Nun gibt es meines Wissens nach keine Funktion zum Kopieren oder Verschieben. Das PSDK erwähnt lediglich die Shell-Funktion "`SHCopyKey`", die aber unter Windows 95 und NT4 den IE5 voraussetzt. Da der Registry-Editor meiner Meinung nach auch schon damals Schlüssel umbenennen konnte, existiert entweder eine undokumentierte Funktion, die das PSDK verschweigt, oder Microsoft macht auch nichts weiter als das, was wir jetzt vorhaben ...

4.6.8.1. Werte enumerieren

Die Enumeration von Schlüsseln haben wir bereits hier besprochen. Schauen wir uns deshalb hier die Funktion "`RegEnumValue`" an, die das selbe für Werte macht. Beim Kopieren eines Schlüssels interessiert uns ja weniger, welche Einträge vorhanden sind, sondern es geht eher um die Frage: sind überhaupt Einträge da?

Wie auch beim Auslesen der einzelnen Schlüssel erwartet die Funktion zunächst das Handle des gerade geöffneten Schlüssels von uns

```
RegEnumValue(src,
```

Es folgt eine null-basierende Variable, die uns als Zähler dient und für jeden Aufruf der Funktion um Eins erhöht wird

```
idx,
```

Danach wird der Puffer angegeben, der den Namen des Eintrags aufnehmen soll

```
ValueName,
```

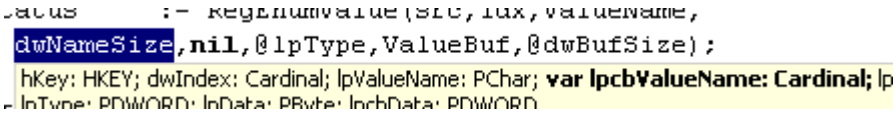
Die Puffergröße (max 16.383 Zeichen) wird auch gleich danach angegeben

```
dwNameSize,
```

Zu beachten ist hier die Deklaration! Laut PSDK ist die Größenangabe ein `LPDWORD`, also ein Zeiger auf einen `dword`-Wert, so dass normalerweise "`@dwNameSize`" anzugeben wäre. Borland hat den Parameter in der "`Windows.pas`" von Delphi 5 jedoch als

```
var lpcbValueName: DWORD;
```

deklariert. Es mag sein, dass das in den neueren (oder auch älteren) Versionen wieder anders ist. Benutzen Sie deshalb im Zweifelsfall die Programmierhilfe von Delphi:



Nach der Größe des Namenspuffers folgt ein reservierter Wert, der nicht verwendet wird.

```
nil,
```

Es folgt der Typ des Registryeintrags, der an die Variable im nächsten Parameter übergeben wird.

```
@lpType,
```

Und da wir natürlich auch am Inhalt des Eintrags interessiert sind, benötigen wir zu guter Letzt einen Puffer, der diesen Inhalt aufnimmt

```
ValueBuf,
```

und dessen Größenwert

```
@dwBufSize);
```

Das ist wieder nur die Kurzform für einen Aufruf. Tatsächlich müssen Sie die Funktion solange aufrufen und dabei die schon erwähnte Zählervariable um Eins erhöhen, bis das Funktionsergebnis `ERROR_NO_MORE_ITEMS` ist. Eine Endlos-Schleife nach dem Muster, das wir beim Aufzählen der Schlüssel verwendet haben, bietet sich also auch hier an.

Und noch eine Ähnlichkeit gibt es: **vor** jedem Aufruf müssen die Variablen mit den Größenwerten des Namens- und des Inhaltspuffers wieder auf ihre Maximalwerte gesetzt werden. Wieso? Nehmen wir als Beispiel den Eintrag "DisplayName". Vor dem Aufruf der Funktion entspricht der Wert von "`dwNameSize`" dem Maximum (16.383), durch die Funktion wird die Variable aber auf den Wert 12 zurückgesetzt (das entspricht den 11 Zeichen inkl. des abschließenden Null-Zeichens). Würden Sie die Funktion nun wieder aufrufen und die Variable nicht zurücksetzen, stünde dem nächsten Eintrag nur ein Puffer von 12 Zeichen zur Verfügung. Hat dieser nächste Eintrag allerdings einen längeren Namen, wäre ein Fehler die Folge, bzw. der Eintrag würde ignoriert werden.
Das gleiche gilt auch für den Puffer, der den Inhalt des Eintrags aufnehmen soll. Der Größenwert, den Sie mit der letzten Variablen übergeben, wird auf die Anzahl der tatsächlich in den Puffer kopierten Bytes gesetzt. Vergessen Sie, diesen Wert wieder auf das Maximum des Puffers zu setzen, steht dem nächsten Eintrag augenscheinlich ein kleinerer Puffer zur Verfügung, was (sofern der Inhalt des nächsten Eintrags größer ist als der des eben gelesenen) den Fehler die Fehlermeldung `ERROR_MORE_DATA` zur Folge hat.

RegEnumValue-Definition

```
LONG RegEnumValue(
  HKEY hKey,              // Handle des geöffneten Schlüssels
  DWORD dwIndex,          // null-basierender Index
  LPTSTR lpValueName,     // Puffer für den Namen des Eintrags
  LPDWORD lpcValueName,   // Größe des Namenspuffers
  LPDWORD lpReserved,     // reserviert (Null)
  LPDWORD lpType,         // Typ des Eintrags
  LPBYTE lpData,          // Puffer für den Inhalt des Eintrags
  LPDWORD lpcbData        // Größe des Inhaltspuffers
);
```

Wenn die Funktion erfolgreich ausgeführt wurde, dann besitzen wir jetzt zum einen den Namen des Eintrags und seinen Inhalt. Beides nutzen wir und legen in unserem neuen Schlüssel eine Kopie davon an. Dazu verwenden wir wie gehabt die Funktion "`RegSetValueEx`", die wir bereits im Kapitel Werte in die Registry schreiben kennen gelernt haben.

4.6.8.2. Schlüssel erzeugen

Es fehlt also noch ein logischer Schritt beim Kopieren eines Schlüssels: seine Unterschlüssel müssen erstens der Reihe nach gesucht und zweitens im Zielschlüssel neu erstellt werden. Wie gesagt, die Enumeration von Schlüssel haben wir uns bereits angesehen. Bleibt also nur noch das Erstellen. Dazu dient die Funktion "`RegCreateKeyEx`".

Gehen wir also im Folgenden davon aus, dass wir gerade dabei sind, die einzelnen Schlüsselnamen mit "`RegEnumKeyEx`" der Reihe nach zu bestimmen. Jeden so ausgelesenen Schlüsselnamen reichen wir dann an "`RegCreateKeyEx`" weiter, wobei wir zuerst das Handle des Zielschlüssels angeben

```
RegCreateKeyEx(dest,
```

Es folgt natürlich der Name des Schlüssels, der im Ziel angelegt werden soll

```
  ValueName,
```

Der nächste Parameter wird wieder einmal nicht verwendet

```
  0,
```

Der nächste Parameter würde auf den Puffer mit den Klassennamen zeigen. Da wir diese aber beim Auslesen auch nicht berücksichtigt haben, können wir sie hier ignorieren

```
  nil,
```

Für den nächsten Parameter stehen drei Konstanten zur Verfügung. Wir verwenden die Vorgabe (Null, oder `REG_OPTION_NON_VOLATILE`)

Wert	Bedeutung
REG_OPTION_NON_VOLATILE	Vorgabe. Der Schlüssel ist "nicht flüchtig" (non volatile). Das heißt, die Informationen werden gespeichert und stehen z.B. nach einem Neustart des Systems wieder zur Verfügung.
REG_OPTION_VOLATILE	Dieses Flag wird unter 9x/ME ignoriert. Hier wird ein Standardschlüssel (non volatile) erzeugt, und die Funktion liefert `ERROR_SUCCESS` als Ergebnis. Unter NT/2000/XP wird mit diesem Flag ein "flüchtiger" (volatile) Schlüssel erzeugt. Die Informationen sind zwar in der Registry sichtbar, werden aber nur im Speicher gehalten und gehen beim Entladen des dazu gehörenden Registryteils verloren. Bei `HKEY_CURRENT_USER` z.B. nach dem Abmelden des aktuellen Benutzers, bei `HKEY_LOCAL_MACHINE` beim Shutdown des Systems. Dieses Flag wird ignoriert, wenn der zu erzeugende Schlüssel bereits existiert.
REG_OPTION_BACKUP_RESTORE	Dieses Flag ist für NT/2000/XP von Interesse. Wenn Sie es nutzen, wird der nächste Parameter (s. Codeausriss) ignoriert. Das Flag erfordert das "SE_BACKUP_NAME"-Recht.

Im nächsten Parameter geben wir die Zugriffsrechte auf den zu erzeugenden Schlüssel an. Da wir davon ausgehen müssen, dass der zu kopierende Schlüssel seinerseits wieder Daten und weitere Unterschlüssel enthält, verwenden wir

```
KEY_READ or KEY_WRITE,
```

Der nächste Parameter spielt für uns keine Rolle

```
nil,
```

Der nächste Parameter ist das Handle auf den neu erzeugten Schlüssel

```
NewTo,
```

und zu guter Letzt ein Parameter, den ich ignoriert habe

```
nil)
```

Wenn Sie ihn verwenden wollen, geben Sie hier einen Zeiger auf eine `dword`-Variable an und prüfen Sie das Ergebnis, das nur zwei Werte annehmen kann:

Wert	Bedeutung
REG_CREATED_NEW_KEY	Der Schlüssel existierte nicht und wurde erzeugt.
REG_OPENED_EXISTING_KEY	Der Schlüssel existierte bereits und wurde einfach nur geöffnet.

RegCreateKeyEx-Definition

```
LONG RegCreateKeyEx(
  HKEY hKey,                                  // Handle des aktuell geöffneten
Schlüssels
  LPCTSTR lpSubKey,                           // Name des Unterschlüssels
  DWORD Reserved,                             // nicht verwendet (Null)
  LPTSTR lpClass,                             // null-terminierter String mit
Klassennamen
  DWORD dwOptions,                            // Speicheroptionen
  REGSAM samDesired,                          // Zugriffsrechte auf den
Schlüssel
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // Sicherheitsattribute (s.
PSDK)
  PHKEY phkResult,                            // Handle des erzeugten
Unterschlüssels
  LPDWORD lpdwDisposition                     // Ergebnis der Aktion
);
```

Das Beispielprogramm enthält eine Prozedur namens "`Reg_MoveKey`" mit allen angesprochenen Aspekten.

5. Grundlagen der GDI

5.1. Grundlagen der GDI

5.1.1. Einführung

Die meisten Zeichenoperation führt Windows mit Hilfe der GDI (Graphics Device Interface) durch. Dazu gehört auch das Zeichnen von Menüs, Schaltflächen, Bildlaufleisten und anderen Oberflächenelementen. Normalerweise merkt man davon nichts, da Windows das alles für uns übernimmt.

Wollen wir aber selbst Objekte (Linien, Rechtecke, gefüllte Flächen, usw.) im Clientbereich unseres Fensters ausgeben, müssen wir auch selbst Hand anlegen. Wie man dabei vorgeht und was zu beachten ist, ist Gegenstand dieses Tutorials.
Da die GDI sehr umfangreich ist (sie enthält immerhin mehrere hundert Funktionen und zugehörige Datentypen), kann dies nur eine oberflächliche Einführung, ein Einstieg, sein.

Die Funktionen, die von der GDI gekapselt werden, befinden sich in der "`GDI32.dll`". Dabei übernimmt sie aber nur die übergeordnete Logik und überlässt die eigentliche Arbeit den jeweiligen Gerätetreiben. Windows kann über den Treiber ermitteln, welche Operationen der Treiber selbst ausführen kann und wo er Unterstützung braucht.
Besitzt die Grafikkarte zum Beispiel einen Co-Prozessor, erhält sie nur die Rohdaten zum Zeichnen. Ist dies nicht der Fall, muss der systemeigene Prozessor diese Arbeit übernehmen; mit Daten, die ihm die GDI32 liefert.

Da der Gerätetreiber also die eigentliche Arbeit übernimmt, kann eine große Geräteunabhängigkeit erzielt werden. Die GDI sagt dem Treiber z.B. nur: "Zeichne eine Linie von Punkt A nach Punkt B". Mehr nicht!
Wie das nun genau auf dem Gerät vonstatten geht (also welche Befehle gesendet werden müssen, damit die Linie auch erscheint), dafür ist allein der Treiber zuständig. Und ob dabei ein Grafikkartentreiber angesprochen wird oder ein Drucktreiber, ist dabei völlig unerheblich.

(Quelle: "Windows-Programmierung", 5. Auflage, Microsoft Press 2000, Charles Petzold)

5.1.2. Grundlagen

Stellen wir uns einmal folgende Situation vor: Auf dem Desktop wird ein Fenster angezeigt. Weil Windows ein Multitasking-Betriebssystem ist, kann es ohne weiteres mehrere Programme/Prozesse mit ihren Fenstern verwalten. So weit so gut. Aber was passiert mit unserem Fenster, wenn ein zweites geöffnet und das erste Fenster dadurch ganz oder teilweise überdeckt wird?

Stellen wir uns den Desktop als großes Bitmap vor. Wird nun das zweite Fenster angezeigt, werden einfach die entsprechenden Pixel geändert, so dass sie das neue Fenster ergeben. Was passiert jetzt aber, wenn das zweite Fenster geschlossen wird? Da müsste ja wieder unserer erstes Fenster sichtbar werden. Damit aber nun kein weißer Fleck zurückbleibt, weist Windows unser Fenster an sich neu zu zeichnen. Die Nachricht, die Windows in diesem Fall schickt, heißt "`WM_PAINT`". Das bedeutet für uns, dass alle Zeichenoperationen im "`WM_PAINT`"-Zweig unserer Fensterprozedur ablaufen müssen; sonst haben wir keine Möglichkeit nach Aufforderung von Windows, unser Fenster zu aktualisieren.

5.1.2.1. GDI-Funktionen

1. Funktionen zum Anfordern oder Freigeben von Gerätekontexten.
   Um zeichnen zu können, muss man sagen wohin gezeichnet werden soll, und das ist unser Gerätekontext. Also praktisch unser Zugang zur Zeichenfläche.
2. Abfragen von Informationen über den Gerätekontext
3. Zeichenfunktionen
4. Funktionen zum Setzen und Abfrage von Attributen
   Unter Attribut wird hier eine Einstellung verstanden, unter der nachfolgende Zeichenoperationen beeinflusst werden; etwa das Setzen der Textfarbe.
5. Funktionen zum Arbeiten mit GDI-Objekten
   GDI-Objekte erzeugt man zum Beispiel mit "`CreatePen`", "`CreatepenIndirect`" usw. Das Prinzip ist dann folgendes: Im Gegensatz zu "`SetTextColor`", welchem man direkt einen Gerätekontext (`DC`, `HDC`) mitgibt,

erzeugen die GDI-Objekte selbst einen DC, den man dann mittels "SelectObject" in den jeweiligen DC einwählt. Ab da werden dann alle Zeichenoperationen mit diesem, so gesetzten Attribut ausgeführt.

5.1.3. Der Gerätekontext

Bevor ein Programm unter Windows zeichnen kann, muss es einen Gerätekontext (DC = "Device Context") für das Ausgabegerät anfordern. Dies stellt sozusagen die Erlaubnis dar, dort zu zeichnen. und es sagt der GDI-Funktion wohin sie zeichnen soll.
Des Weiteren speichert der Gerätekontext eine ganze Reihe von Einstellungen, die man sonst bei jedem Aufruf explizit mit angeben müsste. So kann man sich bei den Funktionsaufrufen auf das Wesentliche beschränken.

5.1.3.1. Ermitteln des Gerätekontextes

Die am häufigsten verwendete Methode zur Ermittlung eines DC besteht in dem Aufruf von "BeginPaint" und "EndPaint" in "WM_PAINT":

```
hdc := BeginPaint(hWnd, ps);
// zeichnen
EndPaint(hWnd, hdc);
```

Als ersten Parameter müssen wir das Handle des Fensters übergeben, dessen DC wir benötigen. Der zweite Parameter ist ein Zeiger auf eine Variable vom Typ TPAINTSTRUCT. Wichtig ist auch das "EndPaint", das Windows mittelt, dass alle Zeichenoperationen abgeschlossen sind und das Fenster aktualisiert wurde. Geben wir Windows keine Rückmeldung, bekommt unser Programm weiter "WM_PAINT"-Nachrichten geschickt, weil Windows der Meinung ist, es müsste sein Fenster noch aktualisieren.

Eine zweite Möglichkeit, einen DC auch außerhalb von "WM_PAINT" zu bekommen, bietet:

```
hdc := GetDC(hWnd);
// zeichnen
ReleaseDC(hWnd, hdc);
```

Das liefert uns einen DC für den Client-Bereich des Fensters. Und mit der folgenden Anweisung erhalten wir den DC auf das komplette Fenster, inkl. Rahmen, Titelleiste, Scrollbars, usw.:

```
hdc := GetWindowDC(hWnd);
// zeichnen
ReleaseDC(hWnd, hdc);
```

5.1.4. Setzen von Eigenschaften

Werfen wir an dieser Stelle einmal einen Blick in den Quellcode und schauen wir uns Schritt für Schritt an was passiert. Zuerst wird ein so genannter Pinsel mit "CreateSolidBrush" erzeugt:

```
RedBrush := CreateSolidBrush(RGB(255, 0, 0));
```

Als einzigen Parameter übergibt man der Funktion einen Farbwert, den man z.B. mit der Funktion "RGB" erzeugen kann. Ergebnis ist ein Handle vom Typ HBRUSH, das wir dann mit "SelectObject in den Gerätekontext einwählen können. Und genau das passiert in der nächsten Zeile:

```
RedBrushOld := SelectObject(WndDC, RedBrush);
```

Diese zwei Parameter sollten klar sein. Der erste gibt den DC an, dessen Eigenschaft wir ändern wollen, und der zweite Parameter kennzeichnet das Objekt, das eingesetzt werden soll. Als Rückgabewert bekommen wir das Handle des zuvor in den DC eingewählten Objektes des gleichen Typs. Dieses sichern wir, um es wieder einzusetzen wenn wir fertig sind. Laut PSDK sollte man nämlich, bevor man das Objekt wieder löscht, das alte einsetzen.

Nun können wir zeichnen. In diesem Fall ein Rechteck, das mit dem aktuell eingestellten Zeichenstift umrandet und mit dem aktuellen Pinsel (Brush) gefüllt ist. Kurz gesagt: ein rotes Rechteck also:

```
Rectangle(WndDC, 80, 10, 100, 90);
```

Verbleiben noch zwei Aktionen: Wiedereinsetzen des alten Objekts (sofern wir es nicht unterwegs in der Hektik verloren haben :o)), und Löschen unseres Objektes, um unserem Programm die Ressourcen wieder zur Verfügung zu stellen

```
SelectObject(WndDc, RedBrushOld);
DeleteObject(RedBrush);
```

Unterlassen wir das, kann der Zeichenspaß ganz schnell ein Ende finden. Denn je nachdem wie aktiv der Benutzer ist, kann es ziemlich häufig dazu kommen, dass Windows unsere Anwendung auffordert, ihr Fenster neu zu zeichnen.

CreateSolidBrush-Definition

```
HBRUSH CreateSolidBrush(
 COLORREF crColor   // brush color value
);
```

5.1.4.1. Muster

Schauen wir uns als kleine Ergänzung noch einmal an, wie man ein anderes Muster erzeugt. Unser obiges, erzeugt mit Hilfe von "`CreateSolidBrush`", war (wie der Name verrät) ein "solides", einfarbiges Muster. Im Prinzip nur eine glatte Fläche. Daneben gibt es natürlich auch die Möglichkeit, ein Muster (im Sinne des Wortes) zu erzeugen. Das Beispielprogramm nutzt dazu die Funktion "`CreateHatchBrush`":

```
GreenHatchBrush := CreateHatchBrush(HS_BDIAGONAL, RGB(0, 255, 0));
```

Das so erzeugte Muster kann nun, nach dem selben Prinzip wie oben beschrieben, zum Füllen des Rechtecks verwendet werden. In diesem Fall besteht das Muster aus diagonalen grünen Linien von links oben nach rechts unten. Das Ändern der Farbe sollte Sie vor keine großen Probleme stellen, und weitere Musterstile (hatch styles) stehen in der Hilfe bzw. im PSDK/MSDN.

CreateHatchBrush-Definition

```
HBRUSH CreateHatchBrush(
  int fnStyle,      // hatch style
  COLORREF clrref   // foreground color
);
```

Das Beispielprogramm erzeugt neben Varianten des Rechtecks ein paar zusätzliche geometrische Figuren.

6. Sonstige Themen

6.1. Subclassing

6.1.1. Was ist Subclassing, und wann braucht man es?

In unserem allerersten Tutorial haben wir gelernt, dass jedes Fenster mit einer Fensterprozedur verknüpft ist. Da es sich bei jedem Control auf unserem Hauptfenster um ein Fenster handelt (ich erwähnte es ja schon), besitzen auch Schaltflächen, Texteingabefenster, Listboxen usw. eine Fensterprozedur. Diese möchte ich als **Standard-Fensterprozedur** bezeichnen.

Diese Standard-Fensterprozedur bekommen wir normalerweise nicht zu sehen, da sie von Windows bereitgestellt wird. Sie ist für das äußere Erscheinungsbild (Darstellung) und für das Verhalten des Controls verantwortlich. Wollen wir nun das Standardverhalten beeinflussen, müssen wir die Standard-Fensterprozedur durch unsere eigene ersetzen und unseren Code implementieren, um das Verhalten des Controls unseren Wünschen anzupassen. In diesem Zug kann man auch gleich noch das Aussehen des Controls beeinflussen.
Letzteres könnte man machen, indem man den Controls den Fensterstil `XX_OWNERDRAW` gibt und sie selbst zeichnet. Dies hat allerdings den Nachteil, dass der Code recht unübersichtlich wird, will man das Aussehen mehrerer Elemente ändern. Mittels Subclassing kann man den Code auslagern und nach Belieben in zukünftige Projekte einbinden. Lediglich die Standard-Fensterprozedur des Controls ist auf die eigene Fensterprozedur "umzubiegen".

Will man das Standardverhalten eines Controls ändern, kommt man um eine eigene Fensterprozedur sowieso nicht herum.

Grob betrachtet ließe sich Subclassing also mit dem Entwerfen eigener VCL-Komponenten vergleichen, die man von einer existierenden ableitet. Da macht man ja auch nichts anderes, als das Standardverhalten und -aussehen seinen Wünschen anzupassen und die bestehenden Methoden und Eigenschaften zu überschreiben.

Demonstrieren möchte ich das Subclassing nun an Hand eines Texteingabefeldes. Ich werde das Aussehen verändern und das Eingabeverhalten so ändern, dass man nur noch Ziffern und Komma eingeben kann.

6.1.2. Die Standard-Fensterprozedur "umbiegen"

Da im Normalfall - wie schon erwähnt - die Standard-Fensterprozedur wirksam wird, müssen wir zunächst dafür sorgen, dass unsere eigene Prozedur aufgerufen werden kann. Um dies zu erreichen, bemühen wir die Funktion "`SetWindowLong`"

```
OldWndProc := Pointer(SetWindowLong(hEdit, GWL_WNDPROC, Integer(@EditWndProc)));
```

die man auch immer dann benutzt, wenn man an einem bestehenden Fenster etwas ändern will; wie z.B. den Fensterstil, die ID, oder (wie hier!) die mit dem Fenster verknüpfte Fensterprozedur. Wir haben diese Funktion bereits bei den "Common Controls" kennen gelernt. Genauer gesagt: bei der Listview, wo wir mit Hilfe der Funktion die Ansicht umgeschaltet haben.

Der erste Parameter ist wie in den meisten Fällen ein Handle auf das betroffene Fenster. Der zweite Parameter ist eine Konstante. Mit dieser Konstante legen wir fest, was geändert werden soll. In unserem Fall ist es deshalb `GWL_WNDPROC`, die Fensterprozedur. Als letzten Parameter geben wir den neuen Wert an; in diesem Fall ist es ein Zeiger auf unsere eigene Fensterprozedur namens "`EditWndProc`".

```
function EditWndProc(hEdit, uMsg, wParam, lParam:DWORD): DWORD; stdcall;
```

Der Rückgabewert ist der vorherige 32-Bit-Wert (Integer), den wir hier casten, da "`CallWindowProc`" einen Zeiger auf eine Fensterprozedur erwartet.

6.1.3. Vom Programmierer zum Maler

Wie jedes gewöhnliche Fenster bekommt unser Eingabefeld die Nachricht "WM_PAINT" wenn es gezeichnet werden muss. Die Standard-Fensterprozedur sorgt normalerweise dafür, dass es wie gewohnt erscheint: vertieft und mit 3D-Effekt. Da nun aber unsere eigene Fensterprozedur etabliert ist, können wir die Nachricht "WM_PAINT" abfangen und das Eingabefeld durch unsere Routinen zeichnen lassen:

```
case uMsg of
  WM_PAINT:
  begin
    { ... }
  end;
end;
```

Machen wir doch mal ein Experiment und kommentieren alles aus, was zwischen **begin** und **end** steht. Und was sehen wir? Wir sehen, dass wir nichts sehen. (Sorry, aber den Kalauer konnte ich mir nicht verkneifen. :o))
Aber warum sehen wir nichts mehr? Weil wir die Nachricht abgefangen haben, allerdings nichts zeichnen. Genau dafür sind wir aber verantwortlich!

Beschäftigen wir uns also ein wenig damit. Der Vergleich mit einem Maler ist gar nicht so weit hergeholt. Wie auch ein Maler haben wir unsere Leinwand, auf die wir zeichnen, und wir haben auch unsere Stifte zum Zeichnen von Linien und unseren Pinsel zum Ausmalen von Flächen.

6.1.3.1. BeginPaint - EndPaint

Als erstes brauchen wir Zugriff auf unsere Leinwand (engl. canvas, und so heißt es auch unter Windows), welche durch unser Fenster repräsentiert wird. Dazu benötigen wir ein Handle auf die Leinwand. In Windows spricht man in diesem Zusammenhang von einem DeviceContext (DC). Unser DC vom Datentyp HDC (in Delphi auch nur ein Ganzzahl-Datentyp) wird uns von der Funktion "BeginPaint" geliefert:

```
dc := BeginPaint(hEdit, ps);
```

Der erste Parameter ist das Handle des Fensters, das von unserer Zeichenaktion betroffen ist. Der zweite Parameter ist ein Zeiger auf eine Variable vom Typ TPAINTSTRUCT. Auf die einzelnen Membervariablen dieses Records möchte ich an dieser Stelle nicht eingehen, da sie für uns nicht von Interesse sind.
Wichtig ist für uns nur der Rückgabewert der Funktion, nämlich unser DC. Außerdem ist noch zu sagen, dass "BeginPaint" das Zeichnen einleitet, während sein Gegenstück "EndPaint" Windows mitteilt, dass die Zeichenaktion abgeschlossen ist.

```
EndPaint(hEdit, ps);
```

Lässt man diese Funktion weg, weiß Windows nichts davon und sendet unserem Fenster weiterhin "WM_PAINT"-Nachrichten. Im ungünstigsten Fall wird unsere Anwendung nur noch diese Nachrichten verarbeiten und kommt zu nichts anderem mehr.

BeginPaint-Definition

```
HDC BeginPaint(
  HWND hwnd,             // handle to window
  LPPAINTSTRUCT lpPaint // paint information
);
```

EndPaint-Definition

```
BOOL EndPaint(
  HWND hWnd,                    // handle to window
  CONST PAINTSTRUCT *lpPaint  // paint data
);
```

tagPAINTSTRUCT-Definition

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc;
  BOOL fErase;
  RECT rcPaint;
  BOOL fRestore;
  BOOL fIncUpdate;
  BYTE rgbReserved[32];
} PAINTSTRUCT, *PPAINTSTRUCT;
```

6.1.3.2. Der Zeichenstift

Genau wie unser Maler müssen auch wir zum Zeichnen einen Stift auswählen:

```
pen := CreatePen(PS_SOLID, 2, RGB(0,0,0));
```

Mit "`CreatePen`" wählen wir einen Stift aus bzw. erzeugen einen. Der Rückgabewert ist ein Handle auf den Stift. Der erste Parameter, "`fnPenstyle`", kann eine der folgenden Konstanten sein:

Wert	Bedeutung
PS_SOLID	Der Stift zeichnet eine durchgehende Linie.
PS_DASH	Der Stift zeichnet gestrichelt.
PS_DOT	Der Stift zeichnet gepunktet.
PS_DASHDOT	Der Stift zeichnet abwechselnd gestrichelt und gepunktet.
PS_DASHDOTDOT	Der Stift zeichnet abwechselnd gestrichelt, gefolgt von zwei Punkten.
PS_NULL	Der Stift ist unsichtbar.
PS_INSIDEFRAME	Dieser Stift zeichnet einen durchgehenden Rahmen und verringert die Maße des Elements, damit es in diesen Rahmen passt.

(Eine ausführlichere Beschreibung finden Sie im MSDN oder im Platform SDK.)

Der zweite Parameter, `nWidth`, ist die Breite des Stiftes in Pixeln, und der letzte Parameter, "`Color`" ist die gewünschte Farbe, die mit der Funktion "`RGB()`" erzeugt werden kann (s. Codebeispiel).

CreatePen-Definition

```
HPEN CreatePen(
  int fnPenStyle,    // pen style
  int nWidth,        // pen width
  COLORREF crColor   // pen color
);
```

Mit "`SelectObject`" nehmen wir den Stift nun in die Hand. Unter Windows spricht man auch von "in den DC selektieren".

```
OldObject := SelectObject(dc, pen);
```

Der erste Parameter ist der DC, in den wir das Objekt selektieren wollen, und der zweite Parameter ist das Handle des Objektes; in dem Fall: unser eben erzeugter Stift. Als Rückgabewert erhalten wir ein Handle des zuvor in den DC selektierten Objekts. Diesen merken wir uns, da wir das gewählte Objekt mit "`DeleteObject`" wieder aus dem DC löschen müssen. Und damit der DC nicht leer zurückgelassen und wieder in den Urzustand versetzt wird, selektieren wir beim Löschen wieder das alte Objekt:

```
DeleteObject(SelectObject(dc, OldObjct));
```

SelectObject-Definition

```
HGDIOBJ SelectObject(
  HDC hdc,          // handle to DC
  HGDIOBJ hgdiobj   // handle to object
);
```

DeleteObject-Definition

```
BOOL DeleteObject(
  HGDIOBJ hObject   // handle to graphic object
);
```

6.1.3.3. Linien zeichnen

Auch wieder wie im richtigen Leben bewegen wir unseren virtuellen Zeichenstift zu dem Punkt, an dem die Linie beginnen soll. Und dann ziehen wir sie bis zu dem Punkt, an dem sie aufhören soll:

```
MoveToEx(dc, rect.Left-2, rect.Top, nil);
LineTo(dc, rect.Left-2, rect.Bottom-7);
LineTo(dc, rect.Right, rect.Bottom-7);
```

6.1.3.4. Rechtecke füllen

Den typischen weißen Hintergrund unseres Eingabefeldes erzeugen wir mit Hilfe folgender Codezeilen:

```
brush := CreateSolidBrush(RGB(255,255,255));
FillRect(dc, rect, brush);
```

Zuerst erzeugen wir einen so genannten "Brush". Wer sich mit den Farbwerten auskennt, sieht schon auf den ersten Blick, dass die Werte für Rot, Blau und Grün am Ende unser gewünschtes Weiß ergeben. Auf diese Weise lassen sich alle beliebigen Farben definieren. Ergebnis ist aber immer ein einfarbiger, solider Brush.
Wer Größeres vorhat, kann sich auch mit "`CreateBrushIndirect`" und eines zugehörigen `LOGBRUSH`-Records austoben. (Näheres dazu bitte im MSDN oder PSDK nachschlagen.)

CreateSolidBrush-Definition

```
HBRUSH CreateSolidBrush(
  COLORREF crColor   // brush color value
);
```

Genau wie der Zeichenstift muss auch der Brush in den DC selektiert werden - eigentlich. In unserem Fall ist dies nicht nötig, da das die Funktion "`FillRect`" für uns übernimmt.

> *Anmerkung*
> Man hätte sich das Erzeugen eines weißen Brushes sparen können und im Aufruf von "`FillRect`" eine Null als dritten Parameter übergeben können - der Null-Brush ist nämlich weiß. (Ich wollte nur mal zeigen, was ich drauf habe. ;o))

FillRect-Definition

```
int FillRect(
  HDC hDC,            // Handle auf den DC
  CONST RECT *lprc,   // das zu füllende Rechteck
  HBRUSH hbr          // Handle des Pinsels, der die Farbe und die Art der
Fläche bestimmt
);
```

Die komplette neue Fensterprozedur finden Sie im Beispielprogramm.

6.1.4. Das Standardverhalten ändern

Wie wir schon beim Zeichnen gesehen haben, bekommt unser Control seine Nachrichten. Diese fangen wir entweder ab oder leiten Sie mit

```
Result := CallWindowProc(OldWndProc, hEdit, uMsg, wParam, lParam);
```

an das System weiter und lassen den Standardcode ausführen. In diesem Kapitel soll nun demonstriert werden, wie man nur bestimmte Zeichen zulässt. Um das zu erreichen, bearbeiten wir die Nachricht "`WM_CHAR`". Das PSDK sagt (frei übersetzt) dazu:

Platform SDK

Die Nachricht **WM_CHAR** wird an das Fenster mit dem Tastaturfokus gesendet, wenn eine **WM_KEYDOWN**-Nachricht von "`TranslateMessage`" bearbeitet wurde. Die Nachricht **WM_CHAR** enthält im `wParam` den Wert Code der gedrückten Taste.

Wir machen also nichts anderes, als alle von uns gewünschten Zeichen an die Standard-Fensterprozedur weiterzuleiten. Nach dem Motto: "Soll sich doch Windows darum kümmern, wie die Zeichen ins Editfeld kommen." Alle anderen Zeichen fallen unter den Tisch, da wir sie überhaupt nicht berücksichtigen:

```
WM_CHAR:
  case Byte(wParam) of
    Byte('0')..Byte('9'),
    Byte(','),
    VK_DELETE,
    VK_BACK:
      CallWindowProc(OldWndProc, hEdit, uMsg, wParam, lParam);
  end;
```

Es ist also recht einfach.

6.2. Ressourcenskripte erstellen

6.2.1. Vorwort

Wer, wie in dem Dialog-Tutorial vorgeschlagen, mit dem Visual Studio oder einem anderen Ressourceneditor arbeitet, wird die eigentlichen Ressourcenskripte nie zu Gesicht bekommen haben. Sie werden aber trotzdem erzeugt, denn ein Ressourcencompiler versteht nur diese Skripte, die die Dateiendung "`.rc`" tragen.
Wem das Visual Studio zu teuer ist und die anderen, billigeren oder kostenlosen Ressourceneditoren zu schlecht oder zu kompliziert sind, wird sich bestimmt schnell mit dem Gedanken anfreunden können, diese Skripte selbst zu schreiben und zu kompilieren.
Ansonsten bleibt einem zumindest das Wissen, wie ein Ressourcenskript aussieht und wie das alles überhaupt funktioniert.

Hinweis

Bei der Arbeit mit den Ressourcenskripts ist ein Platform SDK unabdingbar! Sämtliche Controls und Ressourcen samt ihrer Parameter hier aufzulisten würde exakt dem entsprechen, was das SDK enthält und den Rahmen des Tutorials

bei weitem sprengen. Weitere Ressourcen und Controls werden also im SDK beschrieben, und zwar im Kapitel "Tools and Scripting/SDK Tools/Resource Tools/Resource Compiler".

6.2.2. Einführung in Ressourcenskripte

Ressourcenskripte sind normale Textdateien, die beliebig viele Ressourcendefinitionen und Kommentare enthalten können.
Ein Kommentar wird C-üblich entweder durch doppelte Forward-Slashes (`//`), für einen einzeiligen Kommentar, oder durch einen Forward-Slash, der mit einem Sternchen kombiniert ist (`/* ... */`), für Kommentarblöcke, dargestellt.

Eine Ressourcendefinition hat folgende Syntax:

```
id RESOURCETYPE [Liste von Parametern] [Liste von Statements]
{
  [Control-Definitionen]
}
```

Die `id` ist entweder ein Integer zwischen 0 und 65535 oder ein String, der den Namen der Ressource angibt. Faktisch gesehen handelt es sich bei den Integer-Werten aber auch um Strings, die jedoch in ihre (kürzere) binäre Zeichenform umgewandelt werden (sie verbrauchen daher im Endeffekt weniger Speicher in der fertigen EXE-Datei). Dadurch müssen sie bei ihrer Verwendung mit Hilfe des Makros "`MAKEINTRESOURCE`" entsprechend konvertiert werden; beispielsweise

```
CreateDialog(hInstance,
  MAKEINTRESOURCE(100),  // entspricht '100'
  0,
  @dlgfunc);
```

`RESOURCETYPE` ist einer der 15 Ressourcentypen, u.a. `DIALOG`, `MENU`, `STRINGTABLE` und einige weitere. Hier gilt das gleiche wie beim Identifier. Anstelle des Klarnamens können Sie auch numerische Werte benutzen. Ein gutes Beispiel ist der Wert `24` für das mittlerweile häufig benutzte Manifest von Windows XP. Sinnvoll ist das, wenn Ihr Ressourcencompiler etwas älter ist und daher mit einem Begriff wie `MANIFEST` o.ä. nichts anfangen kann.

Die Liste der Parameter ist abhängig vom Typ der Ressource. Bei Dialogen hat man die Position und die Höhe und Breite des Fensters, beim Bitmap nur den Dateinamen.

Die Liste der Statements ist nicht bei jeder Ressource und auch nicht immer vollständig verfügbar. Welcher Ressourcentyp welche Statements unterstützt, steht im SDK. Einiges ist aber auch logisch, so macht bei einem Bitmap ein `CAPTION`-Statement nicht sehr viel Sinn und wird deswegen nicht erlaubt. Auch wenn man alle Statements in einer Zeile hintereinander schreiben kann, erhöht es doch die Lesbarkeit erheblich, wenn man nach jedem Statement jeweils eine neue Zeile anfängt. Den Compiler stört das überhaupt nicht.

Die Control-Definitionen sind immer in geschweiften Klammern (alternativ auch `BEGIN` und `END`) eingeschlossen und können beliebig viele weitere Definitionen enthalten. Jede Definition geht bis zum Ende einer Zeile, dementsprechend kann auch immer nur eine Definition pro Zeile stehen. Einige Ressourcen, wie z.B. `ICON` oder `BITMAP`, akzeptieren jedoch keine Controls.
Eine Control-Definition wiederum hat folgende Syntax:

```
CONTROLTYPE [text,] id, x, y, width, height [, weitere Parameter]
```

`CONTROLTYPE` gibt den Typ des Controls an, z.B. `LTEXT` für einen statischen, linksbündigen Text oder `PUSHBUTTON` für einen Button.
`text` gibt die Caption des Controls an. Der Text muss in doppelten Anführungszeichen stehen und entsprechend der C-Syntax escaped, also Sonderzeichen für den Compiler entsprechend markiert, werden. Beispielsweise muss ein "`\`" also entsprechend "`\\`" geschrieben werden. Ein Kaufmannsund (&) vor einem Buchstaben kennzeichnet diesen als "Mnemonic", also als Tastenkürzel. Der Buchstabe wird automatisch unterstrichen. Beachten Sie bitte, dass nicht jedes Control eine Caption hat!

Die `id` ist der Identifier des Controls, der bei Notify-Messages zur Identifikation übergeben wird oder bei Funktionen wie

"SetDlgItemText" übergeben werden muss.

x, y, width und height geben die Position und die Maße des Controls an. Diese werden nicht in Pixeln, sondern in Dialog Units angegeben, die ich im nächsten Kapitel erklären werde. Beachten sie auch hier, dass es Ausnahmen gibt, die keine Position- und Größenangaben akzeptieren, wie z.B. Strings aus einer STRINGTABLE-Ressource.

Die weiteren Parameter sind optional und Control-abhängig. Die meisten unterstützen eine HelpID oder verschiedene Styles (die übrigens genauso definiert sind wie die Windows-Styles).

6.2.3. Ressourcenskripte kompilieren

Es gibt viele verschiedene Ressourcencompiler. Hat man Delphi, hat man schon einen davon: den **brcc32** von Borland. Er befindet sich im "bin"-Ordner des Delphi-Verzeichnisses und benötigt die DLL "rw32core.dll", wenn man ihn von einem anderen Ort aus ausführen will. Damit man nicht für jedes Projekt eine eigene Kopie des Compilers mit herumschleppt, bietet es sich an den Pfad des Compilers in die PATH-Variable in der "autoexec.bat" einzutragen.

Ein typischer Aufruf des brcc32 sieht folgendermaßen aus:

```
brcc32 filename.rc
```

Das kompiliert die Datei "filename.rc" (im aktuellen Verzeichnis) und erzeugt eine Datei "filename.res" im gleichen Verzeichnis wie die RC-Datei.
Um eine andere Zieldatei anzugeben, kann man brcc32 mit dem Parameter "-foFILENAME" aufrufen:

```
brcc32 filename.rc -foc:\somepath\somefile.res
```

Die Ressourcendatei wird jetzt nicht mehr im aktuellen Verzeichnis mit dem Namen des Scripts gespeichert, sondern im Verzeichnis "c:\somepath\" mit dem Dateinamen "somefile.res".

Andere Kommandozeilenparameter erhält man mit

```
brcc32 -?
```

6.2.4. Dialog-Units

Dialog Units sind abhängig von der Schriftart, die dem Dialog zugewiesen ist. Das bringt einen Vorteil, aber auch einen riesigen Nachteil. Der Vorteil ist die Unabhängigkeit von festen Bezugspunkten. Um einen Dialog zu vergrößern, muss man nur dessen Schriftart vergrößern, alle Controls passen sich an. So sind auch Dialoge auf Systemen mit speziellen Zugangshilfen für Sehbehinderte bei den großen Schriften nirgendwo abgeschnitten. Das ist aber auch gleichzeitig der Nachteil. Bei der Entwicklung von Dialog-Ressourcen muss man oft stundenlang rumprobieren, bis das Layout den Vorstellungen entspricht und zwischendurch natürlich ständig neu kompilieren, falls man den unten beschriebenen Trick nicht kennt.
Dialog Units sind, wie schon gesagt, von der Schriftart und -größe des Dialoges abhängig. Eine horizontale Dialog Unit ist ein Viertel der durchschnittlichen Zeichenbreite in Pixeln, eine vertikale Dialog Unit entspricht einem Achtel der durchschnittlichen Zeichenhöhe.

Doch nun zum erwähnten Trick: Wer schlecht im Schätzen ist oder sich gar nicht erst die Mühe machen will, durch Probieren ans Ziel zu gelangen, kann sich das Leben mit Taschenrechner und einem kleinen Testprogramm vereinfachen.
Dazu muss man zunächst einen leeren Dialog mit der gewünschten Schriftart zu erzeugen. Das RC dazu sieht wie folgt aus:

```
// Dialog1
100 DIALOGEX 0, 0, 162, 95
STYLE DS_3DLOOK | DS_NOFAILCREATE | DS_CENTER | WS_MINIMIZEBOX |
  WS_MAXIMIZEBOX | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Arbeiten mit Dialog-Ressourcen"
FONT 8, "MS Sans Serif"
{
}
```

Das ist das Skript aus dem Dialog-Tutorial, jedoch wurden alle Control-Definitionen weggelassen (also entsteht ein leeres Fenster).
Als Font wurde MS Sans Serif mit 8 Punkten angegeben, dies ist die Dialogschrift.

Jetzt kompiliert man die Ressource und schreibt ein kleines Programm, das die Ressource lädt.
Die Funktion "`MapDialogRect`" rechnet Dialog Units in Pixel um und übernimmt als Parameter das Handle zum Dialog und einen TRect-Record mit den Werten, die umgerechnet werden sollen.
Um die Anzahl an Pixeln für eine Dialog Unit zu bekommen, sieht der Aufruf wie folgt aus:

```
var
  DialogUnits: TRect = (Left: 1; Top: 1);

  // [...]
  // (z.B. in der Bearbeitung für die WM_INITDIALOG-Message)
  MapDialogRect(hDlg, DialogUnits);
  // [...]
```

`DialogUnits` enthält jetzt in `Left` und `Top` jeweils die Anzahl an Pixel für eine horizontale, bzw. vertikale Dialog Unit. Da aber die Werte nicht immer gleichmäßig sind, weil bei Dezimalzahlen gerundet wird, sollte man nicht nur für 1 prüfen, sondern bei mehreren verschiedenen Werten und dann den Durchschnitt bilden. Kein Problem, wenn das Ergebnis eine Dezimalzahl sein sollte: da Windows rundet, können Sie das getrost auch tun.
Mit diesem einfachen Trick kann man sich das Leben sehr vereinfachen, jetzt kann man nämlich wie gewohnt in Pixeln arbeiten und braucht vorher nur umzurechnen, den entsprechenden Faktor hat man ja nun.

MapDialogRect-Definition

```
BOOL MapDialogRect(
    HWND hDlg,     // Handle to a dialog box.
    LPRECT lpRect  // Pointer to a RECT structure that contains the dialog box
coordinates to be converted.
);
```

6.2.5. Der erste Dialog

Fangen wir nun an mit unserem echten ersten Dialog.

Wenn wir im PSDK unter DIALOG nachschlagen, sehen wir, dass als optionale (aber dennoch relevante) Statements `CAPTION`, `CLASS`, `EXSTYLE`, `FONT`, `LANGUAGE`, `MENU` und `STYLE` verfügbar sind. Auf die drei etwas exotischeren werde ich später genauer eingehen. Ein einfacher Dialog sähe also folgendermaßen aus:

```
100 DIALOG 0, 0, 200, 100
STYLE WS_VISIBLE | WS_CAPTION | WS_SYSMENU | WS_MAXIMIZEBOX |
  WS_MINIMIZEBOX | DS_NOFAILCREATE | DS_CENTER
CAPTION "Ein einfacher Dialog"
FONT 8, "MS Sans Serif"
{
  CTEXT "Ein Text als Beispiel", 101, 1, 3, 197, 11, WS_BORDER
  PUSHBUTTON "Klick mich", 102, 74, 50, 50, 16
}
```

Lädt man diesen Dialog, hat man ein normales Fenster mit einem zentrierten, umrandeten Text und einem Button darauf. An das Fenster werden natürlich die Messages der Children geschickt, daher kann man in der "DlgProc" ohne Weiteres "WM_COMMAND" abfangen und auf die BN_CLICKED-Notification des Buttons reagieren (die ID ist 102). Hier unterscheiden sich die selbst gestrickten Dialoge in keiner Weise von den Resultaten eines Dialogeditors.
Wir sehen auch, dass die Styles des Dialoges (genauso wie alle anderen verknüpfbaren Ausdrücke auch) mit einem bitweisen **ODER** verknüpft werden - allerdings in der C-Schreibweise als Pipe (|) - zu erhalten mit AltGr + <.
Alle *TEXT-Controls sind mit normalen STATICs vergleichbar, die aus dem Label-Tutorial bekannt sind. Das Sternchen steht hier für drei Buchstaben: L (linksbündig), R (rechtsbündig) und C (zentriert).

6.2.5.1. Exoten

Die drei Statements eines Dialoges, die nicht besonders häufig verwendet werden und auch nicht immer Verwendung finden, sind CLASS, LANGUAGE und MENU.

Das CLASS-Statement ermöglicht es, dem Dialog eine bestimmte Fensterklasse zuzuweisen. Diese muss natürlich im Programm, das die Ressource lädt, vorher definiert und mit RegisterClass im System registriert werden. Der Parameter des Statements ist ein String, der in doppelten Anführungszeichen steht:

```
100 DIALOG 0, 0, 200, 150
CLASS "TestClass"
```

Definiert wird also ein Dialog, dessen Klassenname "TestClass" lautet. Die Zuweisung der Fensterklasse muss jetzt wie folgt aussehen:

```
wc: TWndClassEx = (
  lpszClassName   : 'TestClass';
  cbWndExtra      : DLGWINDOWEXTRA;
  // weitere Klasseneigenschaften
  // [...]
);
```

Was auffällt ist, dass cbWndExtra der Wert DLGWINDOWEXTRA zugewiesen wird. Das hat den Grund, dass ein Dialog zusätzliche Informationen speichert und somit mehr Speicher benötigt. Mit

```
cbWndExtra := DLGWINDOWEXTRA
```

reservieren wir eben diesen zusätzlichen Speicher, und zwar vollkommen automatisch. Würden wir den Wert auf 0 lassen, würde es zu Abstürzen kommen. Macht Ihr Programm also plötzlich Sachen, die es nicht machen sollte, überprüfen Sie zuerst, ob die Klasse korrekt definiert ist.
Diese Art von Subclassing ermöglicht es auch, auf einfachste Art und Weise eine echte "WndProc" für einen Dialog zu erzeugen (und so auch "WM_CREATE" abzufangen).

Das LANGUAGE-Statement dient dazu, Ressourcen für mehrere Sprachen anzubieten. Die Ressource hat den gleichen Identifier, aber eine unterschiedliche Sprachsignatur. Eine solche Signatur setzt sich aus einem primären und einem sekundären Language-Identifier zusammen, z.B.

```
100 DIALOG 0, 0, 200, 100
STYLE WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_NOFAILCREATE
CAPTION "Ein einfacher Dialog"
FONT 8, "MS Sans Serif"
LANGUAGE LANG_GERMAN, SUBLANG_GERMAN_SWISS
{
}
```

für einen deutschen Dialog mit schweizerischen Sondermerkmalen. Eine vollständige Liste aller Identifiers findet sich im SDK.

Das `MENU`-Statement dient dazu, einem Dialog ein Menü zuzuweisen. Das Laden und Zuweisen zur Laufzeit entfällt also. Als Parameter nimmt das `MENU`-Statement den Identifier (entweder als Zahl oder als String ohne Anführungszeichen) des Menüs an:

```
100 MENU // oder "hello MENU" als String-Identifier
{
  MENUITEM "&Some menu item", 101
}

200 DIALOG 0, 0, 200, 100
STYLE WS_VISIBLE | WS_CAPTION | WS_SYSMENU | DS_NOFAILCREATE
CAPTION "Ein einfacher Dialog"
FONT 8, "MS Sans Serif"
MENU 100 // oder "MENU hello" als String-Identifer
{
}
```

Menü-Ressourcen werde ich noch eingehender beschreiben.

Das war eine kleine Einführung in eine einfache Dialog-Ressource. Dialog-Ressourcen können genauso komplex werden wie "normale" Fenster. Allerdings gibt es in der Komplexität der Dialog-Ressourcen in der Win9x-Reihe bis einschließlich Windows ME die Einschränkung von 255 Controls pro Dialog. Braucht man mehr, empfiehlt das SDK die restlichen Controls in der "`WM_INITDIALOG`"-Message zu erzeugen.

6.2.6. Menüs ganz einfach

Ein sehr beliebter Anwendungsbereich für Ressourcen sind Menüs. Menüs sind meist statisch, die sich ändernden Einträge werden entweder deaktiviert oder nachträglich ausgeblendet. Diese Möglichkeit zur fast beliebigen nachträglichen Bearbeitung ist eine ideale Voraussetzung, um zumindest das Grundgerüst des Menüs in eine Ressource zu packen, anstatt jeden Eintrag einzeln hinzuzufügen.

Der Ressourcentyp dazu ist `MENU` (Vorsicht: nicht verwechseln mit dem Statement `MENU`!). Das einzig relevante Statement einer `MENU`-Ressource ist `LANGUAGE`, das die gleiche Bedeutung wie bei den Dialogen hat.
Eine `MENU`-Ressource besteht aus zwei Arten von Items: `MENUITEM`-Controls und `POPUP`-Controls. `MENUITEM`s sind einfache Menüeinträge, `POPUP`s sind aufklappbare Menüs die wiederum selbst `MENUITEM`s und `POPUP`s beinhalten können.

Eine Ressource für das Search-Menü aus Delphi würde wie folgt aussehen:

```
100 MENU
{
  POPUP "&Search"
  {
    MENUITEM "&Find...", 101
    MENUITEM "Fin&d in Files...", 102
    MENUITEM "&Replace...", 103
    MENUITEM "&Search Again", 104
    MENUITEM "&Incremental Search", 105
    MENUITEM SEPARATOR
    MENUITEM "&Go to Line Number...", 106
    MENUITEM "Find &Error", 107
  }
}
```

Dieses Menü kann man jetzt wie gewohnt mit LoadMenu laden und mit SetMenu einem Fenster zuweisen.

Wie gesagt, eine `MENU`-Ressource kann mehrere Einträge besitzen, genauso wie ein `POPUP`-Control selbst auch `POPUP`s aufnehmen kann. Ein komplexeres Menü sähe also so aus:

```
100 MENU
{
  POPUP "&Search"
  {
    MENUITEM "&Find...", 101
    MENUITEM "Fin&d in Files...", 102
    MENUITEM "&Replace...", 103
    MENUITEM "&Search Again", 104
    MENUITEM "&Incremental Search", 105
    MENUITEM SEPARATOR
    MENUITEM "&Go to Line Number...", 106
    MENUITEM "Find &Error", 107
  }

  POPUP "&zweites Menü"
  {
    MENUITEM "ein schöner &Eintrag", 110
    POPUP "&und sogar ein Untermenü"
    {
      MENUITEM "da ist der &Untereintrag", 120
    }
  }
}
```

Was man allerdings immer noch selber machen muss ist, den Menüeinträgen Bitmaps und Tastenkürzel zuzuweisen. Dafür gibt es in Menü-Ressourcen (noch) keine Möglichkeit.

6.2.7. Bitmaps, Icons, Cursor

Die einfachsten (aber nicht weniger wertvollen) Ressourcen sind solche, die binäre Dateien einfügen. Sie nehmen als einzigen Parameter den Dateinamen, eingeschlossen in doppelten Anführungszeichen, an.
Bitmaps, Icons und Cursors unterscheiden sich untereinander nur darin, dass vor dem Linken der Dateien eine Prüfung der Datei vorgenommen wird. So wird verhindert, dass man ein normales Bitmap als Cursor einbindet.
Folgendes Script bindet mehrere Dateien ein:

```
100 CURSOR "pinsel.cur"
200 BITMAP "Strandbild.bmp"
300 BITMAP "Winterbild.bmp"
400 ICON "BildKonvertieren.ico"
```

Diese Ressourcen können mit den Funktionen "LoadCursor", "LoadBitmap" und "LoadIcon" geladen werden, allerdings empfiehlt das SDK die Benutzung der flexibleren Funktion LoadImage:

```
myCursor := LoadImage(hInstance, MAKEINTRESSOURCE(100),
  IMAGE_CURSOR, 0, 0, LR_VGACOLORS);
```

Übrigens ermöglicht es "LoadImage" im Gegensatz zu "LoadIcon" auch, eine bestimmte Icon-Größe anzugeben, falls die Icon-Datei mehrere Icons enthält.

6.2.8. RCDATA und binäre Ressourcen

Ein kleiner Sonderfall ist die RCDATA-Ressource. Ihr Parameter kann beliebige Dateien aufnehmen, es wird also keine Dateityperkennung durchgeführt. Aber zusätzlich hat man die Möglichkeit, binäre Daten direkt in das Ressourcescript einzugeben. So kann man ganze Datenstrukturen direkt in die Ressourcen speichern:

```
100 RCDATA "somefile.exe"
200 RCDATA
{
  123  // WORD
  456L // LONGWORD
}
```

Die resultierende Ressourcendatei enthält 2 Ressourcen. Einmal die Datei "somefile.exe", die dann geladen, temporär gespeichert und ausgeführt werden kann.

Die zweite Ressource ist eine binäre Ressource, deren Daten direkt im Script festgelegt wurden. Sie ist 6 Bytes lang (Word + LongWord), wobei die ersten 2 Bytes die Zahl 123 enthalten und die restlichen 4 die Zahl 456. Diese binäre Struktur kann man jetzt direkt in einen Record laden, aber man muss beachten, dass der Ressourcencompiler keinerlei Ausrichtung der Daten übernimmt. Die Deklaration eines solchen Record muss also unbedingt **packed** sein:

```
TResRec = packed record
  Val1: Word;
  Val2: LongWord;
end;
PResRec = ^TResRec;
```

Bei dieser Art, binäre Daten in eine Ressource zu packen, muss man allerdings ein paar Dinge beachten

- Strings werden, wie in C üblich, in doppelten Anführungszeichen geschrieben, Zeichen die entsprechend dem ANSI-C-Standard escaped werden müssen, werden durch einen Backslash escaped.
- Da der Ressource-Compiler die Daten in keiner Weise verändert, sollte man bei Strings ein Nullzeichen ("Some String\0") anhängen, um standardkonform zu bleiben.
- Ein L hinter einer Zahl deklariert diese Zahl eindeutig als LongWord, ansonsten ist jede Zahl ein Word.
- Hexadezimale Zahlen werden mit "0x" angeführt, oktale mit "0o".
- Ein L vor einem String definiert ihn als Wide-String (UNICODE). Alle anderen Strings sind ANSI-Strings.

Bei Strings sollte man zusätzlich bedenken, dass der String als "array of Char" in der Ressource liegt, nicht als PChar und auch nicht als normaler String. Entsprechend muss man die Länge des Strings vorher wissen und ein entsprechend langes Array verwenden. Für Sprachressourcen, deren Strings je nach Sprache verschiedene Längen haben, ist diese Methode also ungeeignet.

RCDATA-Ressourcen lädt man mit den Befehlen "FindResource", "LoadResource" und "LockResource". Erstere liefert einen speziellen *Resource Location Block* zurück, den "LoadResource" dazu verwendet, die Daten zu laden. "LockResource" schließlich hält die Daten im Speicher fest und liefert einen Pointer auf den Datenbereich.

FindResource-Definition

```
HRSRC FindResource(
    HMODULE hModule, // Handle to the module that contains the resource
    LPCTSTR lpName,  // Name of the resource
    LPCTSTR lpType   // Specifies the type of the resource
);
```

LoadResource-Definition

```
HGLOBAL LoadResource(
    HMODULE hModule, // Handle to the module that contains the resource
    HRSRC hResInfo   // Handle to the resource to be loaded
);
```

LockResource-Definition

```
LPVOID LockResource(
    HGLOBAL hResData // Handle to the resource data to be locked
);
```

Für unsere Ressource sähe das also folgendermaßen aus:

```
var
  hMod: hModule;
  hRes: HRSRC
  hData: hGlobal;
  Data: PResRec;
begin
  hMod  := LoadLibrary(PChar(paramstr(0));
  hRes  := FindResource(hMod, MAKEINTRESOURCE(200), RT_RCDATA);
  hData := LoadResource(hMod, hRes);
  Data  := PResRec(LockResource(hData));

  { ... }
end;
```

Data zeigt jetzt auf unsere zwei Integer, die wir nun beliebig verwenden können.

6.2.9. Stringtables

Stringtables speichern eine Reihe von Strings, die man einzeln laden kann. Die Stringtable an sich ist aber keine Ressource, sondern nur ein virtueller Container für die eigentlichen Strings und sieht folgendermaßen aus:

```
STRINGTABLE
{
  100 "Hello, world"
  200 "Ein sehr langer String \
der auf mehrere Zeilen verteilt ist"
  300 "Ein anderer String \012der in der nächsten Zeile
weitergeht"
}
```

Die Stringtable besteht also aus einer Liste Strings, die in doppelten Anführungszeichen steht. Der Backslash am Ende einer Zeile (es dürfen auch keine Leerzeichen folgen) dient dazu, lange Strings auf mehrere Zeilen zu verteilen. Die Zeichenfolge `\012` steht für eine neue Zeile.
Ein NULL-Terminator ist nicht erforderlich, weil er durch `LoadString` eingefügt wird, womit wir auch schon beim Laden solcher String-Ressourcen sind:

```
var
  buf: PChar;
begin
  GetMem(buf, 100);
  LoadString(hInstance, MAKEINTRESSOURCE(100), buf, 100);

  { ... }
end;
```

Die Variable buf enthält jetzt den String mit dem Identifier 100. `LoadString` fügt automatisch den NULL-Terminator an den String an.

LoadString-Definition

```
int LoadString(
    HINSTANCE hInstance, // Handle to the instance that contains the resource
    UINT uID,            // Name of the resource
    LPTSTR lpBuffer,     // Buffer that receives the string
    int nBufferMax       // Size of the buffer
);
```

6.2.10. Versionsinformationen

Zum Abschluss werfen wir noch einen Blick auf einen Block mit Versionsinformationen. Sie kennen diese Angaben, wenn Sie sich die Eigenschaften einer Datei ansehen. Enthalten ist meist das Copyright, der Firmenname, der interne Name, usw.
Eine Versionsresource trägt den Bezeichner `VERSIONINFO` und sieht wie folgt aus

```
VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,0
 PRODUCTVERSION 1,0,0,0
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x10004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040704b0"
        BEGIN
            VALUE "CompanyName", "Win32-API-Tutorials\0"
            VALUE "FileDescription", "Beschreibung\0"
            VALUE "FileVersion", "1.0.0.0\0"
            VALUE "InternalName", "Interner Name\0"
            VALUE "LegalCopyright", "Copyright © 2004 Mathias Simmack.\0"
            VALUE "OriginalFilename", "Originaler Dateiname\0"
            VALUE "ProductName", "Produkt\0"
            VALUE "ProductVersion", "1,0,0,0\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x407, 1200
    END
END
```

Die Angaben für `FILEFLAGS`, `FILEOS`, `FILETYPE` und `FILESUBTYPE` richten sich nach bestimmten Werten, die Sie in der Headerdatei "`WinVer.h`" des PSDK finden können.

So sehen Sie bspw., dass die Angabe `FILEFLAGS` im Normalfall Null ist. Wenn die Compilerdirektive `_DEBUG` gesetzt wird, dann erhält das Attribut den Wert Eins (`VS_FF_DEBUG`) - das Debug-Flag wird gesetzt. Ähnlich verhält es sich auch mit den anderen Angaben. Der Wert `$10004` in `FILEOS` steht z.B. für eine Win32-Anwendung.

Aufmerksam machen möchte ich Sie auf die beiden Angaben

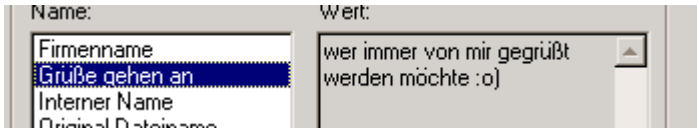
```
 FILEVERSION 1,0,0,0
 PRODUCTVERSION 1,0,0,0
```

Das Komma ist kein Fehler, weil es sich bei diesen beiden Angaben um 32-Bit-Werte handelt. Jede Stelle ist also eins der vier Bytes. Im Gegensatz dazu handelt es sich hier um Strings:

```
            VALUE "FileVersion", "1.0.0.0\0"
            VALUE "ProductVersion", "1,0,0,0\0"
```

Es wäre also in dem Fall unerheblich, ob Sie zur Trennung der einzelnen Stellen einen Punkt oder ein Komma verwenden. Nur dürfen Sie bei all diesen Strings (s. auch oben) nicht die abschließende Null vergessen, die ich in dem Auszug fett markiert dargestellt habe.
Eine Regel gibt es für diesen Stringblock in dem Sinn eigentlich nicht. Die oben gezeigten Werte sind üblich, Sie könnten aber auch eigene Strings definieren, die dann natürlich auch angezeigt werden:

```
            VALUE "Grüße gehen an", "...\0"
```



6.2.11. Versionsinformationen im Programm auslesen

Zum Auslesen von Versioninformationen benötigen wir ein paar Funktionen aus dem API. Zum einen müssen wir herausfinden, wie groß diese Ressource ist. Dazu benutzen wir "`GetFileVersionInfoSize`", die den entsprechenden Wert zurückgibt.
Wenn der Wert größer als Null ist, reservieren wir entsprechend Speicher und laden den Versionsblock mit der Funktion "`GetFileVersionInfo`"

```
vis           := GetFileVersionInfoSize(pchar(FileName),dummy);
if(vis > 0) then
begin
  GetMem(vi,vis);
  try
    GetFileVersionInfo(pchar(Filename),0,vis,vi);
    if(vi = nil) then exit;
```

Wie Sie in der o.g. Beispielressource sehen können, befinden sich die gesuchten Strings in einem Block namens "StringFileInfo". Darunter befindet sich ein weiterer Block namens "040704b0", dessen Name allerdings variieren kann. Aus dem Grund muss vorher der zweite Block, "VarFileInfo", ausgelesen werden, dessen "Translation"-Wert mit dem o.g. Block identisch ist. Der "Translation"-Wert ist ein `longint`, dessen zwei Stellen (`word`) Sie im Beispiel sehen können.
Um den Wert auszulesen, verwenden wir "`VerQueryValue`", die zum einen den Zeiger auf den Speicher benötigt, gefolgt von der Angabe des gewünschten Blocks, dem Zielzeiger als dritten Parameter und einer Variablen für die Größe ganz am Schluss

```
    VerQueryValue(vi,'\\VarFileInfo\\Translation',translation,vis);
    if(translation = nil) then exit;
```

Der Zeiger "`translation`" zeigt nun auf den Wert. Um auf den Blocknamen (im Beispiel "040704b0") zu kommen, reicht es aus, den `longint` hexadezimal formatiert zu verwenden. Der Einfachheit halber kann man dazu "`Format`" benutzen. Auf die Weise sparen wir auch das Hantieren mit weiteren Variablen, weil wir den gesuchten Stringnamen gleich anhängen:

```
    VerQueryValue(vi,
      pchar(Format('\\StringFileInfo\\%.4x%.4x\\%s',
        [LOWORD(longint(translation^)),HIWORD(longint(translation^)),
        BlockKey])),ip,vis);
    if(ip = nil) then exit;

    SetString(Result,pchar(ip),vis);
  finally
    FreeMem(vi);
  end;
end;
```

Das Ergebnis ist die Funktion "`GetFileInfo`", die Sie zum Auslesen aller Stringwerte des "StringFileInfo"-Blocks verwenden können; bspw. für die Beschreibung:

```
SetDlgItemText(hDlg,CID_LABEL1,
  pchar(GetFileInfo(paramstr(0),'FileDescription')));
```

6.2.12. Der feste Versionsblock

Dabei handelt es sich um die Werte am Anfang,

```
 FILEVERSION 1,0,0,0
 PRODUCTVERSION 1,0,0,0
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x10004L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
```

Diese müssen separat abgefragt werden, was allerdings auch etwas einfacher wird, da es dafür ein spezielles Record mit allen Daten gibt, auf die man einfacher zugreifen kann. Die Grundfunktion sieht auch ähnlich aus, nur dass diesmal die beiden Blöcke keine Rolle spielen, sondern dass wir nach dem Reservieren des Speichers und dem Laden der Versionsressource auf den Hauptblock zugreifen

```
VerQueryValue(vi,'\\',pointer(FixBuf),dummy);
if(FixBuf = nil) then exit;
```

"`FixBuf`" ist ein Zeiger auf ein Record vom Typ TVSFixedFileInfo. Dieses Record enthält alle gezeigten Werte. Um bspw. die Versionsnummer als Ergebnis zu erhalten, genügt die folgende Formatierung:

```
Result := Format(FormatStr,
 [(FixBuf^.dwFileVersionMS and $FFFF0000) shr 16,
   FixBuf^.dwFileVersionMS and $0000FFFF,
  (FixBuf^.dwFileVersionLS and $FFFF0000) shr 16,
   FixBuf^.dwFileVersionLS and $0000FFFF]);
```

6.3. Anwendung für die Systemsteuerung

6.3.1. Vorwort

Eine Anwendung für die Systemsteuerung zu schreiben ist gar nicht so schwer. Die Grundlage dafür kann nämlich eins Ihrer Programme bilden. Idealerweise ist dies vielleicht ein Programm mit Dialogen oder so genannten "Property Sheets". Das läuft letztlich auf das selbe hinaus, da auch Property Sheets mit Dialogressourcen arbeiten. Zu beachten sind lediglich die beiden folgenden Dinge:

1. Ein Systemsteuerungsmodul ist eine DLL mit der Endung "`.cpl`".
2. Es muss die Funktion "`CPlApplet`" exportiert werden, weil hierüber die Kommunikation zwischen System und der Anwendung abläuft.

Um Ihnen zu zeigen, dass es tatsächlich recht einfach ist, habe ich als Beispiel das Programm aus dem Dialog-Tutorial benutzt und nur ein bisschen ergänzt.

Da von der Seite der Fenster, Dialoge und Controls nicht wirklich etwas neues hinzukommt, wird sich dieser Beitrag ganz auf die Funktionen der Systemsteuerung konzentrieren. Sprich: auf die Kommunikation zwischen dem System und Ihrer Anwendung. Meine Empfehlung lautet daher, dass Sie sich die entsprechenden Grundlagenbeiträge noch einmal ansehen, wenn Sie etwa mit Dialogen, Buttons o.ä. Schwierigkeiten haben.

6.3.2. Grundlegende Änderungen am Code

Da ich bereits schrieb, dass eine Anwendung für die Systemsteuerung eigentlich eine DLL ist, müssen wir selbstverständlich den Programmkopf "`program`" durch "`library`" ersetzen. Außerdem benötigen wir die Unit

"Cpl.pas", in der speziell CPL-Nachrichten und -Records enthalten sind, mit deren Hilfe wir die Anwendung später dazu bringen, korrekt auf die Anfragen des Systems zu reagieren

```
library ControlPanel;

uses
  { ... },
  Cpl;
```

Wenn Sie min. Delphi 4 besitzen, können Sie auch bereits im Programmcode die Endung der kompilierten Datei festlegen. Als Systemsteuerungsmodul muss unser Programm die Endung ".cpl" besitzen

```
{$E cpl}
```

Ihre Dialogprozeduren u.ä. können Sie im Moment unverändert übernehmen. Wichtig ist nur die neue Funktion "CPlApplet", die jedes Systemsteuerungsmodul haben muss. Auf die Funktion selbst komme ich im nächsten Kapitel zurück, hier soll nur erwähnt werden, dass der Name "CPlApplet" kein Schreib- oder Tippfehler ist. Das große P ist Absicht und muss auch von Ihnen benutzt werden, sonst wird die Anwendung später in der Systemsteuerung nicht angezeigt

```
function CPlApplet(hwndCpl: HWND; uMsg: UINT; lp1, lp2: LPARAM): longint;
  stdcall;
begin
  { ... }
end;
```

Diese Funktion muss von der Anwendung exportiert werden

```
exports
  CPlApplet;
```

Die einzige Möglichkeit, intern einen anderen Funktionsnamen als "CPlApplet" zu verwenden, steht Ihnen durch die Benutzung von "name" zur Verfügung. Damit legen Sie den zu exportierenden Funktionsnamen selbst fest, bspw.

```
exports
  MyCplFunc name 'CPlApplet';
```

Und da die Kommunikation diesmal anders abläuft, sollten Sie nach Möglichkeit auch den Hauptteil des Programms leeren. Da Sie hier sicher im Normalfall den notwendigen Code zum Erzeugen und Anzeigen der Dialoge o.ä. aufrufen, müssen Sie sich Gedanken darüber machen, wie Sie dies in einer Anwendung für die Systemsteuerung handhaben wollen. Speziell der Code für den Dialog muss natürlich entfernt werden, da er erst später bei der Auswahl des Moduls in der Systemsteuerung angezeigt werden soll. Gegen den Aufruf von Prozeduren und Funktionen, mit denen Sie grundlegende Dinge initialisieren, ist aber natürlich nichts zu sagen.

```
begin
end.
```

So weit, so gut.

6.3.3. Die Funktion "CPlApplet"

Die Funktion "CPlApplet" dient, wie bereits in den vorigen Kapiteln erwähnt, als Vermittler zwischen der Anwendung und dem System. Das System sendet Nachrichten an die Anwendung, auf die Sie reagieren müssen. Ich möchte Ihnen hier die grundlegende Kommunikation schrittweise demonstrieren. Das bedeutet, der hier gezeigte Code ist also Teil der Funktion "CPlApplet"

```
function CPlApplet(hwndCpl: HWND; uMsg: UINT; lp1, lp2: LPARAM):
  longint; stdcall;
begin
  Result := 1;

  case uMsg of
    // Nachrichtenbearbeitung
    { ... }

    else
      Result := 0;
  end;
end;
```

Und wie Sie sehen können gibt es eigentlich auch keine großen Unterschiede zur bisher bekannten Nachrichtenbearbeitung von Fenstern und Dialogen.
Doch weiter im Text.

1. Zuerst empfängt Ihre Anwendung die Nachricht "CPL_INIT". Das ist das Signal dafür, dass die Systemsteuerung aufgerufen wurde und sich nun bemüht, die einzelnen Module zu laden. Sie haben hier die Möglichkeit, weiteren Code zur Initialisierung Ihrer Anwendung zu schreiben. Wichtig ist, dass Sie mit dem Rückgabewert Eins antworten, wenn Ihre Anwendung geladen werden soll. Geben Sie Null als Ergebnis der Nachricht zurück, beendet das System die Kommunikation mit Ihrem Programm und lädt es auch nicht.
   Da in meinem Beispiel Eins der Standardwert ist, reicht es hier aus, die Property Sheets (dazu im nächsten Punkt) zu initialisieren:

   ```
   CPL_INIT:
     InitPropertySheet;
   ```

2. Die zweite Nachricht ist "CPL_GETCOUNT". Hier geben Sie als Reaktion die Anzahl der in Ihrer Anwendung enthaltenen Einzelmodule zurück. Das hört sich verwirrend an? Ist es aber nicht. Ihre Anwendung kann nämlich aus mehreren Teilen bestehen. Das Beispiel etwa besitzt neben dem Originaldialog aus dem Dialog-Tutorial einen zweiten Dialog und ein Property Sheet. Diese drei "Module" sollen nun separat in der Systemsteuerung angezeigt werden, und darum muss das Rückgabeergebnis in dem Fall auch 3 lauten.
   Allerdings können Sie es sich auch einfacher machen. Dazu muss ich ein wenig auf den nächsten Punkt vorgreifen. Ich habe ein Record mit folgendem Aufbau definiert:

   ```
   type
     CPL_MODULES = record
       icon,                       // Ressourcen-ID des Symbols
       name,                       // Ressourcen-ID des Modulnamens (String-
   ID)
       description,                // Ressourcen-ID der Beschreibung (String-
   ID)
       dlgtemplate : integer;      // Ressourcen-ID des Dialogs
       dlgproc     : pointer;      // Zeiger auf Dialogprozedur
     end;
   ```

   Dieses Record findet sich in der Variablen "CplModules" wieder, die als Array deklariert ist

   ```
   var
     CplModules : array[0..2]of CPL_MODULES;
   ```

   Kurz gesagt kann ich daher die Größe dieses Arrays als Ergebnis auf die Nachricht "CPL_GETCOUNT" benutzen:

   ```
   CPL_GETCOUNT:
     Result := length(CplModules);
   ```

3. Die nächste Nachricht, "CPL_INQUIRE", ist von der Anzahl der Einzelmodule abhängig. Das bedeutet, dass sie entsprechend oft aufgerufen wird. Bei unseren drei Modulen also dreimal.
   Sie müssen hier das Symbol, den Namen und eine erklärende Beschreibung für jedes Modul zurückliefern. Und genau da kommt das o.g. Record ins Spiel. Wie Sie sehen können enthält es integer-Werte für die drei

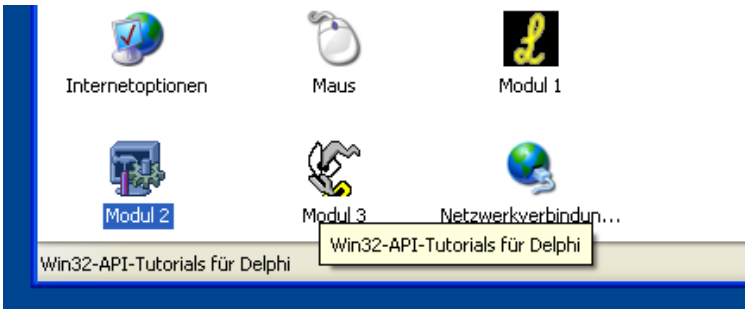
genannten Eigenschaften. Das heißt, Sie müssen an der Stelle also die entsprechenden Ressourcen-IDs angeben. Welches Modul gerade abgefragt wird, das erfahren Sie mit Hilfe von `lp1` aus dem Funktionskopf. Dieser Zähler ist null-basierend und entspricht damit auch dem Bereich der Variable "`CplModules`".
Nun müssen die Daten aber noch irgendwie an das System übermittelt werden. Hier ist `lp2` (aus dem Funktionskopf) wichtig. Diese Variable enthält einen Zeiger auf ein Record vom Typ "`TCplInfo`", das wir mit unseren Werten füllen müssen. Dank der o.g. Variable geht das recht einfach und schnell:

```
CPL_INQUIRE:
  begin
    PCplInfo(lp2)^.idIcon := CplModules[lp1].icon;
    PCplInfo(lp2)^.idName := CplModules[lp1].name;
    PCplInfo(lp2)^.idInfo := CplModules[lp1].description;
    PCplInfo(lp2)^.lData  := 0;
  end;
```

TCplInfo-Definition

```
typedef struct tagCPLINFO {
    int idIcon;
    int idName;
    int idInfo;
    LONG_PTR lpData;
} CPLINFO
```

Wenn bis hier alles geklappt hat, dann müsste die Systemsteuerung nun auch Ihre drei neuen Module anzeigen.



Wenn der Anwender eins auswählt, wird die Nachricht "`CPL_DBLCLK`" gesendet. In dieser Nachricht sorgen Sie dann dafür, dass die jeweiligen Dialoge angezeigt werden. Auch hier wird in der Variablen `lp1` der Index des gewünschten Moduls übermittelt, so dass im Fall meines Beispiels lediglich geprüft werden muss, ob dieser Wert 2 ist. Dann soll nämlich der Property Sheet-Dialog angezeigt werden, was aber einen anderen Befehl voraussetzt. In allen anderen Fällen kann ich "`DialogBox`" benutzen und mit Hilfe von "`CplModules`" die richtige Dialogressource und die richtige Dialogprozedur laden

```
CPL_DBLCLK:
  if(lp1 = 2) then PropertySheet(psh)
    else DialogBox(hInstance,MAKEINTRESOURCE(CplModules[lp1].dlgtemplate),
      hwndCpl,CplModules[lp1].dlgproc);
```

Prinzipiell reicht das bereits aus, um eine funktionierende Anwendung für die Systemsteuerung zu erstellen. Es gibt aber noch zwei weitere wichtige Nachrichten, auf die Sie im Bedarfsfall reagieren können. Etwa, um belegte Ressourcen o.ä. freizugeben. Diese Nachrichten sind

Wert	Bedeutung

CPL_STOP	Diese Nachricht wird gesendet, bevor die Systemsteuerung die geladenenen CPL-Dateien wieder freigibt. Sie ist von der Anzahl der in der CPL-Datei enthaltenen Module abhängig und wird entsprechend oft gesendet.
CPL_EXIT	Wird nach "`CPL_STOP`" und vor dem Freigeben der Systemsteuerung gesendet.

6.3.4. Die CPL-Datei registrieren

Ein sehr wichtiges Thema, meiner Ansicht nach, ist das Registrieren der CPL-Datei. Zwar sollte der Dateityp im System bekannt sein, so dass ein Doppelklick im Explorer auch bereits den Dialog öffnet, aber wenn Sie das mit der Beispielanwendung probieren, dann werden Sie immer nur den ersten Dialog (= das erste Einzelmodul) sehen. Aus dem Grund sollten Sie bei der Weitergabe Ihrer CPL-Datei ein Setup oder ein Hilfsprogramm verwenden, mit dem sie im System registriert werden kann. Dabei gibt es allerdings Betriebssystem-bedingte Unterschiede zu beachten.

Windows 9x und NT

Eine Möglichkeit wäre, die CPL-Datei direkt in das Systemverzeichnis von Windows zu kopieren. Dadurch würde sie automatisch gefunden und geladen werden. Allerdings wird dieser Weg generell von Microsoft missbilligt. Sie sollten es nach Möglichkeit vermeiden, eigene Dateien in wichtige System- o.ä. Ordner zu kopieren. Das gilt nicht nur für CPL-Anwendungen sondern generell für diverse Dateien.

Der bessere Weg ist, die CPL-Datei in der "`control.ini`" zu registrieren. Erstellen Sie in dieser Datei ggf. die Sektion `MMCPL` und tragen Sie hier Pfad und Namen Ihrer CPL-Datei ein. Auf die Weise kann sie im Ordner Ihrer Anwendung bleiben und steht der Systemsteuerung dennoch zur Verfügung

```
[MMCPL]
TUTTESTCPL=C:\Programme\Ihre Anwendung\CONTRO~1.CPL
```

Wählen Sie nach Möglichkeit einen kurzen Dateinamen, oder benutzen Sie das im Beispiel gezeigte Muster, denn zumindest Windows 98 hat ein Problem mit CPL-Dateien, deren Namen das Maß 8.3 übersteigen. Interessanterweise gilt das aber nicht für den Pfad.

Hinweis

> Weil diese Art der Registrierung auch für Windows NT 4 gilt, sollten Sie sich Gedanken über die Benutzerrechte machen. Ob Sie die CPL-Datei nun in das Systemverzeichnis kopieren, oder ob Sie die genannte INI-Datei bearbeiten, beides setzt meiner Ansicht nach min. Admin-Rechte voraus. Ihr Setup bzw. Ihr Registrationsprogramm sollte dies entsprechend berücksichtigen.

Windows 2000, XP und höher

Ab Windows 2000 tragen Sie den Namen der CPL-Datei in die Registry ein. Dabei spielen die Benutzerrechte natürlich eine wichtige Rolle. Wenn Sie möchten, dass die CPL-Datei allen Benutzern zur Verfügung steht, dann müssen Sie im Schlüssel `HKEY_LOCAL_MACHINE` tätig werden, was allerdings Administratorrechte erfordert. Mit normalen Benutzerrechten können Sie die CPL-Datei aber auch unter `HKEY_CURRENT_USER` eintragen, womit sie dann aber auch nur dem gerade angemeldeten Benutzer zur Verfügung steht. Der Pfad ist in beiden Fällen identisch:

```
Software\Microsoft\Windows\CurrentVersion\Control Panel\Cpls
```

Erstellen Sie hier einen String vom Typ `REG_EXPAND_SZ`, dem Sie als Wert den Pfad und Namen der CPL-Datei übergeben, bspw.

```
RegSetValueEx(reg,'TUTTESTCPL',0,REG_EXPAND_SZ,
  pointer(ExtractFilePath(paramstr(0)) + 'ControlPanel.cpl'),
  length(ExtractFilePath(paramstr(0)) + 'ControlPanel.cpl'));
```

Schauen Sie sich ggf. den Beitrag über die Registry an. Ich werde an dieser Stelle nicht weiter darauf eingehen, wie man sie öffnet und Werte liest oder schreibt.

Windows XP Kategorien

Wenn Sie Windows XP besitzen, dann ist Ihnen sicher schon die neue Kategorienansicht der Systemsteuerung aufgefallen. Dabei handelt es sich um eine Art logische Gruppierung, bei der die einzelnen Module thematisch zusammengefasst werden. Sie können Ihre eigene CPL-Datei ebenfalls in diese Kategorien aufnehmen. Allerdings sind dazu min. Administratorrechte erforderlich, weil der notwendige Eintrag unter `HKEY_LOCAL_MACHINE` vorgenommen werden muss.

Zuerst muss man wissen, welche Kategorien es gibt. Informationen darüber finden Sie im PSDK, im Kapitel "User Interface Services/Windows Shell/Shell Programmer's Guide/What's New in the Shell/Control Panel Categories".

Kategorie	Bedeutung
Appearance and Themes	entspricht Darstellung und Designs, also den Modulen zur optischen Anpassung von Windows
Printers and Hardware	Hardware-Assistent, Drucker, Maus, Scanner, usw.
Network and Internet	Netzwerkeinstellungen, Internetoptionen
Sound, Speech and Audio Devices	das, was man unter dem Begriff Multimedia zusammenfassen würde
Performance and Maintenance	Systemverwaltung ("Leistung und Wartung")
Date, Time, Language, ...	Datum, Zeit, regionale Einstellungen, usw.
Accessibility	Eingabehilfen

Außerdem gibt es noch zwei besondere Kategorien

Kategorie	Bedeutung
Add or Remove Programs	das Software-Modul mit den installierten Programmen
User Accounts	Benutzerkonten

Diese beiden öffnen normalerweise separate Fenster, bspw. die Liste mit den installierten Programmen. Wenn Sie allerdings eigene Module integrieren, dann verhalten sie sich wie die anderen Kategorien und zeigen eine Auswahl an. Das sich normalerweise selbstständig öffnende Fenster (die Liste mit der Software, um beim Beispiel zu bleiben) sehen Sie dann nur, wenn Sie das entsprechende Symbol auswählen.

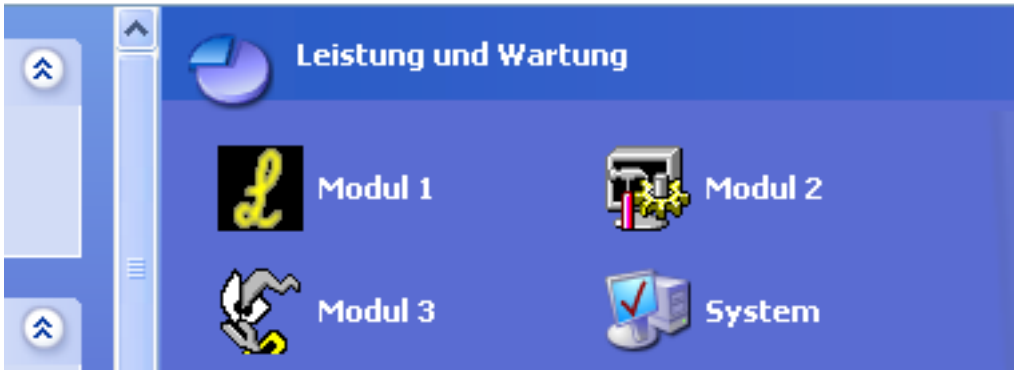
Wie gesagt, Admin-Rechte sind erforderlich, und der Pfad lautet:

```
SOFTWARE\Microsoft\Windows\CurrentVersion\Control Panel\Extended Properties
```

Hier finden Sie einen Unterschlüssel namens "`{305CA226-D286-468e-B848-2B2E8E697B74} 2`" (die 2 am Ende ist kein Fehler!), der die notwendigen Einträge in Form von `dwords` enthält. Für mein Beispiel verwende ich die Kategorie "Leistung und Wartung", die den Wert `0x05` hat. Der folgende Code stammt aus dem Programm "`RegisterCPL.dpr`", das der Beispielanwendung beiliegt

```
RegSetValueEx(reg,
  pchar(ExtractFilePath(paramstr(0)) + 'ControlPanel.cpl'),
  0,
  REG_DWORD,
  @XPCAT_MAINTENANCE,
  sizeof(dword));
```

Sie erstellen also lediglich einen `dword`-Schlüssel mit dem Namen Ihrer CPL-Datei und ordnen diesem den Wert der gewünschten Kategorie zu. Und das Ergebnis kann sich sehen lassen:



Allerdings sehen Sie auch, dass Sie u.U. mehrere CPL-Dateien erstellen müssen, wenn Sie Module für verschiedene Kategorien entworfen haben. Da Sie lediglich den Dateinamen im o.g. Schlüssel angeben, werden alle in der CPL-Datei enthaltenen Einzelmodule der selben Kategorie zugeordnet.

> **Apropos: Windows XP**
> Wenn Sie Ihren Modulen das Aussehen von XP (sprich: die Themes) spendieren wollen, dann muss Ihre Manifestressource zwingend die ID `123` haben. Bei allen anderen Werten wird die Ressource ignoriert und der jeweilige Dialog im alten, klassischen Stil angezeigt.
>
> ```
> 123 RT_MANIFEST "../../Common~1/manifest.xml"
> ```

6.4. Einen Assistenten erstellen

6.4.1. Was ist ein Assistent?

Als Assistent bezeichnet man üblicherweise ein Programm, das aus mehreren Dialogseiten besteht, zwischen denen man beliebig wechseln kann. Beendet wird ein Assistent meist durch eine Schaltfläche mit der Aufschrift "Fertig stellen" o.ä. Gesehen haben Sie so ein Programm sicher schon. Einige sind im Betriebssystem eingebaut, manche Setupprogramme arbeiten nach diesem Prinzip, usw. usw.

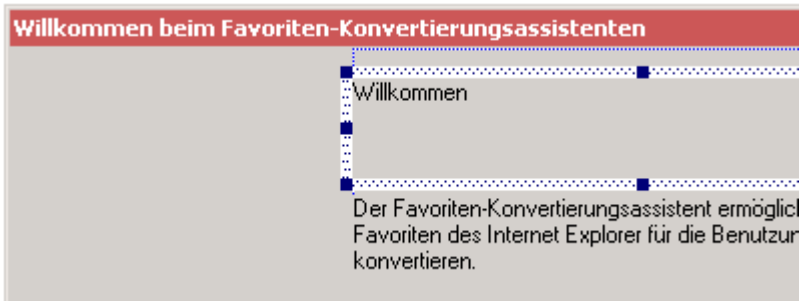
Von der Umsetzung her entspricht ein Assistent einem so genannten "Property-Sheet"-Control. Nun stellt man sich aber darunter eher eine Seite mit mehreren Registern vor. Jedes Register entspricht einem eigenen Dialog. Der Anwender kann auf das Register klicken und sieht dann den entsprechenden Dialog, in dem er Eingaben tätigen, Optionen auswählen und dergleichen mehr tun kann.
Was hat das aber mit einem Assistenten zu tun? Die Grundlage ist die gleiche. Nur zeigt der Assistent eben nicht alle Seiten auf einmal an, sondern man kann die Optionen nur nacheinander erreichen. Glücklicherweise ist der Code dank Property-Sheets im System integriert, so dass es nur weniger Handgriffe bedarf, um einen Assistenten zu erzeugen.

Wir wollen als Beispiel für einen Assistenten einen IE-Favoritenkonverter für den Opera schreiben. Sie wissen sicher, dass der Internet Explorer eine eingebaute Import- und Exportfunktion für die Favoriten besitzt. Allerdings wird beim Export lediglich die Mozilla-typische "`bookmarks.html`" unterstützt. Ein Problem ist das freilich nicht, denn der Opera kann

diese Datei natürlich importieren. Aber wir wollen den Kreis schließen und den Export in das vom Opera benutzte Format direkt anbieten.
Allerdings werden Sie beim Lesen feststellen, dass es eigentlich keine Rolle spielt, was das Beispielprogramm macht. Es geht ja auch hauptsächlich darum, wie man ein typisches Assistentenprogramm erstellt. Dass das Beispielprogramm nicht nur als Demonstration der Möglichkeiten anzusehen ist, ist sicher ein angenehmer Nebeneffekt. :o)

6.4.2. Dialogseiten erstellen

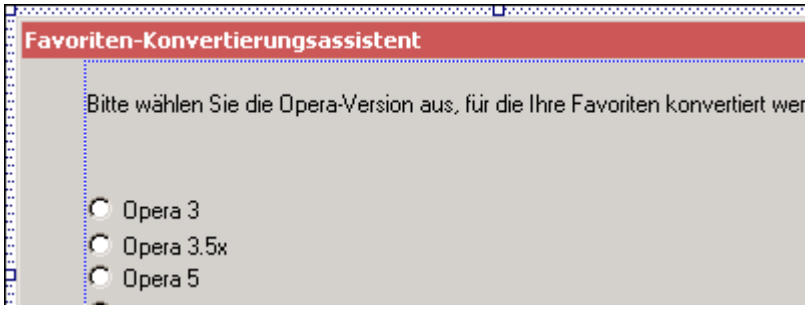
Die Grundlage eines Assistenten bilden eine oder mehrere Dialogseiten, die nacheinander angezeigt werden. Das folgende Bild stammt aus dem Visual Studio 6. Ich habe den oberen Text absichtlich markiert, damit Sie sehen, dass es sich um zwei Controls handelt:



Das hat damit zu tun, dass der Text im oberen Feld später vergrößert dargestellt werden soll, damit er als Überschrift erkennbar ist. Die Dialogressource des Programms enthält insgesamt vier solcher Dialogseiten. Davon haben allerdings nur die Begrüßungs- und Beendenseite eine Höhe von 193 Dialogunits, weil sie links eine so genannte "Watermark Bitmap" anzeigen. Die beiden anderen Seiten haben stattdessen einen Header und benutzen darum nur 143 Dialogunits als Höhe. Die Breite ist allerdings bei allen Dialogseiten identisch: 317 Dialogunits.
Diese Maße entsprechen den Vorgaben von Microsoft, die Sie im PSDK im Kapitel "User Interface Services/Wizard 97" finden.

Dass die Dialoge Titelzeilen haben, ist kein Fehler. Natürlich werden die Seiten später im eigentlichen Assistentenfenster "eingebettet", aber der Text aus den Titelzeilen wird dennoch angezeigt.

Ein Unterschied muss noch deutlich gemacht werden: wie Sie im Bild oben sehen, beginnt der Text ziemlich weit rechts. Das hat mit der schon erwähnten Bitmap zu tun, die am linken Rand angezeigt werden soll. Zwar stellt das System diese Funktionalität zur Verfügung, d.h. Sie müssen die Grafik lediglich angeben, da sie aber Teil Ihres Dialoges wird, müssen Sie ihre Breite natürlich berücksichtigen.
Im Gegensatz dazu zeigt das folgende Bild eine der beiden inneren Seiten. Die besitzen eine geringere Höhe (das habe ich auch schon erwähnt) und benutzen keine Bitmap am linken Rand, so dass man den dortigen Platz natürlich auch nutzen kann:



6.4.3. Die Dialogseiten laden

Wie bei einem herkömmlichen Programm, das Dialogressourcen nutzt, müssen die einzelnen Seiten natürlich auch hier geladen werden. Die Reihenfolge, in der die Seiten später im Assistenten angezeigt werden, ergibt sich dabei aus der Reihenfolge, in der Sie sie laden lassen.

Zum Laden der Seiten benötigt man ein `TPropSheetPage`-Record. Dessen Eigenschaften entscheiden u.a. auch über das Aussehen der jeweiligen Seite. Doch fangen wir vorn an. Das Record muss zuerst mit seiner Größe initialisiert werden

```
psp.dwSize             := sizeof(psp);
```

Danach wird die `dwFlags`-Membervariable mit Werten gefüllt,

Wert	Bedeutung
`PSP_DEFAULT`	Der Wert ist Null und bedeutet laut PSDK: "uses the default meaning for all structure members". Ich gebe zu, ich habe keine Ahnung, wie man das am sinnvollsten übersetzen soll. :o)
`PSP_HIDEHEADER`	Die Seite besitzt keinen Header.
`PSP_HASHELP`	Es wird ein Hilfe-Button angezeigt, wenn die Seite sichtbar ist.
`PSP_USEHEADERTITLE`	Die Seite besitzt einen Titel im Header; üblicherweise ist dieser Text fett gedruckt.
`PSP_USEHEADERSUBTITLE`	Die Seite besitzt zusätzlichen Text im Header; üblicherweise unter dem Header in normaler Schrift, dafür aber leicht eingerückt.

(weitere Werte finden Sie im PSDK)

Für unsere erste Dialogseite benutzen wir nur die folgenden beiden Flags:

```
psp.dwFlags            := PSP_DEFAULT or PSP_HIDEHEADER;
```

Für eine so genannte "innere Seite", die einen Header anzeigen soll, sind anstelle von `PSP_HIDEHEADER` `PSP_USEHEADERTITLE` und ggf. `PSP_USEHEADERSUBTITLE` zu benutzen

```
psp.dwFlags            := PSP_DEFAULT or PSP_USEHEADERTITLE or
  PSP_USEHEADERSUBTITLE;
```

wobei dann natürlich auch die entsprechenden Strings zu benutzen sind. Im Fall des Beispielprogramms werden diese aus den Ressourcen geladen:

```
psp.pszHeaderTitle     := MAKEINTRESOURCE(IDS_P1TITLE);
psp.pszHeaderSubTitle  := MAKEINTRESOURCE(IDS_P1SUBTITLE);
```

Zu guter Letzt werden die Anwendungsinstanz, eine Dialogprozedur und natürlich auch die Dialogressource selbst gebraucht:

```
psp.hInstance          := hInstance;
psp.pfnDlgProc         := @IntroDlgProc;
psp.pszTemplate        := MAKEINTRESOURCE(IDD_INTRODLG);
```

Das bedeutet, dass Sie nach Möglichkeit nicht die gleiche Dialogprozedur für verschiedene Seiten benutzen sollten. Das mag sich umständlich anhören, ist aber meiner Ansicht nach letztlich auch übersichtlicher.
Nun kommt der Zusammenhang von Assistenten und Property-Sheet, denn die eigentliche Seite wird mit dem Befehl "`CreatePropertySheetPage`" erzeugt. Rückgabewert ist das Handle der neu erzeugten Seite, sofern die Aktion erfolgreich war. Das Interessante hierbei ist allerdings, dass wir das Handle in einem Array

```
var
  ahpsp : array[0..3]of HPROPSHEETPAGE;
```

sichern:

```
ahpsp[0]               := CreatePropertySheetPage(psp);
```

Das liegt daran, dass unser Programm ja aus mehreren Seiten besteht, die auch alle angezeigt werden sollen.

TPropSheetPage-Definition

```
typedef struct _PROPSHEETPAGE {
    DWORD dwSize;                   // Recordgröße
    DWORD dwFlags;                  // Flags
    HINSTANCE hInstance;            // Anwendungsinstanz
    union {
        LPCSTR pszTemplate;         // Dialogressource
        LPCDLGTEMPLATE pResource;
        };
    union {
        HICON hIcon;
        LPCSTR pszIcon;
        };
    LPCSTR pszTitle;                // alternativer Titel
                                    // (benötigt PSP_USETITLE-Flag)
    DLGPROC pfnDlgProc;             // Dialogprozedur
    LPARAM lParam;
    LPFNPSPCALLBACK pfnCallback;
    UINT *pcRefParent;
#if (_WIN32_IE >= 0x0500)
    LPCTSTR pszHeaderTitle;         // Haupttitel des Headers
    LPCTSTR pszHeaderSubTitle;      // untergeordneter Titel des Headers
#endif
#if (_WIN32_WINNT >= 0x0501)
    HANDLE hActCtx;
#endif
}
```

6.4.4. Den Assistenten erzeugen

Nachdem alle Dialogseiten geladen wurden, kann der Container für den Assistenten erzeugt werden. Das System übernimmt, wie bereits erwähnt, die grundlegenden Funktionen. Wir benötigen lediglich ein weiteres Record, "`TPropSheetHeader`", das natürlich zuerst wieder initialisiert werden muss:

```
ZeroMemory(@psh,sizeof(psh));
psh.dwSize              := sizeof(psh);
psh.hInstance           := hInstance;
```

Als übergeordnetes Fenster benutzen wir in diesem Fall Null (den Desktop). Sie könnten hier auch das Handle Ihrer Anwendung benutzen, wenn Sie den Assistenten darüber starten:

```
psh.hwndParent          := 0;
```

Als nächstes weisen wir die zuvor geladenen Dialogseiten zu. Hierfür benötigen wir das Handle-Array aus dem letzten Kapitel. Es muss der Membervariablen `phpage` übergeben werden, die ein Pointer ist. Darum müssen wir direkt das erste Element des Array referenzieren:

```
psh.phpage              := @ahpsp[0];
```

Dann geben wir an, welche Seite unsere Startseite ist (null-basierender Index), und wie viele Seiten es insgesamt gibt:

```
psh.nStartPage          := 0;
psh.nPages              := 4;
```

Die `dwFlags`-Membervariable habe ich nicht grundlos an das Ende gestellt. Hier geben Sie nämlich an, welche zusätzlichen Eigenschaften benötigt werden. Einige davon sind beim Assistenten allerdings nicht notwendig, weil sie nur bei den echten Property-Sheets Sinn machen. Für ein Assistenten-artige Aussehen mit Header und Bitmaps sind die drei folgenden Stilattribute zu setzen:

```
psh.dwFlags             := PSH_WIZARD97 or PSH_WATERMARK or PSH_HEADER;
```

Auf Besonderheiten und andere Flags komme ich sofort zurück, doch im Moment ist das Laden der Bitmaps von Interesse. Wie bereits angekündigt soll unser Assistent auf der Begrüßungs- und Beendenseite eine Bitmap am linken Rand anzeigen. Dazu dient das Flag `PSDH_WATERMARK`. Und `PSH_HEADER` bedeutet, dass wir auch in den "inneren" Seiten eine Bitmap anzeigen wollen - allerdings rechts oben, im Header.
Die Bitmaps lassen sich über zwei Wege laden. Sie könnten die `hbmWatermark`- und `hbmHeader`-Variablen benutzen. Diese erwarten eine, bspw. mit "`LoadBitmap`" zuvor geladene Grafik. Da sich die Bitmaps ja aber sowieso in den Ressourcen befinden, ist das Laden mit den beiden Membervariablen `pszbmWatermark` und `pszbmHeader` einfacher. Hier geben Sie nämlich nur den Namen der beiden Ressourcen an:

```
psh.pszbmWatermark      := MAKEINTRESOURCE(IDB_WATERMARKBMP);
psh.pszbmHeader         := MAKEINTRESOURCE(IDB_HEADERBMP);
```

Es bleibt eigentlich nur noch eins zu tun: wir müssen den Assistenten anzeigen. Dazu reicht der Aufruf des Befehls "`PropertySheet`", dem wir das `TPropSheetHeader`-Record als Parameter übergeben:

```
PropertySheet(psh);
```

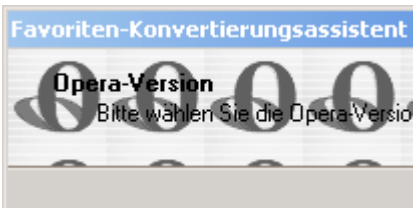
Den Rest übernimmt das System. - Halt! So ganz vollständig ist die Sache natürlich noch nicht. Es fehlen noch die Dialogprozeduren, auf die aber im nächsten Kapitel eingegangen werden wird. Darum können wir uns jetzt noch ein paar Attribute anschauen:

Wert	Bedeutung
`PSH_WIZARD`	Erzeugt einen Assistenten **ohne** Header und ohne die Möglichkeit, links eine Begrüßungsgrafik anzuzeigen.
`PSH_WIZARD97`	Erzeugt einen moderner wirkenden Assistenten, der Header und eine Bitmap am linken Rand enthalten kann. *
`PSH_WIZARDCONTEXTHELP`	Zeigt den Kontexthilfe-Button in der Titelleiste an.
`PSH_WIZARDHASFINISH`	Zeigt den "Fertig stellen"-Button bei jeder Seite an.
`PSH_HASHELP`	Zeigt einen deaktivierten Hilfe-Button an. Der Button wird nur bei Seiten aktiviert, die das PSP_HASHELP-Attribut benutzen.

Achtung!
Wenn es bei Ihnen zu dem Phänomen kommt, dass die Bitmaps gekachelt werden, dann liegt dies mit hoher Wahrscheinlichkeit an einem falschen bzw. veralteten Wert für `PSH_WIZARD97`. Da der Funktionsumfang von der Bibliothek "`comctl32.dll`" abhängig ist, die durch den Internet Explorer aktualisiert werden kann, gibt es einen Wert für Systeme mit einem IE kleiner als Version 5 und einen Wert für Systeme mit dem IE>=5. Im PSDK sieht die Definition von `PSH_WIZARD97` wie folgt aus:

```
#if (_WIN32_IE < 0x0500)
#define PSH_WIZARD97            0x00002000
#else
#define PSH_WIZARD97            0x01000000
#endif
```

Wenn Sie Delphi 5 besitzen, dann werden Sie mit diesem Problem zu tun haben, da die "`CommCtrl.pas`"-Unit nur den alten Wert (IE<5) enthält, wodurch folgendes entsteht:



Ich habe daher in der, dem Beispielprogramm beiliegenden Unit "`CommCtrl_Fragment.pas`" den neuen Wert und

weitere neue Werte definiert, damit Sie obigen Darstellungsfehler verhindern und auch gleichzeitig auf neuere Attribute u.ä. zugreifen können. Der alte Wert ist in dieser Unit als `PSH_WIZARD97_IE4` gespeichert.

TPropSheetHeader-Definition

```
typedef struct _PROPSHEETHEADER {
    DWORD dwSize;                       // Recordgröße
    DWORD dwFlags;                      // Flags
    HWND  hwndParent;           // Elternfenster
    HINSTANCE hInstance;                // Anwendungsinstanz
    union {
        HICON hIcon;            // Symbol
        LPCTSTR pszIcon;                // nur bei Property-Sheets
        };
    LPCTSTR pszCaption;
    UINT nPages;                        // Anzahl der Seiten
    union {
        UINT nStartPage;                // Startseite
        LPCTSTR pStartPage;
        };
    union {
        LPCPROPSHEETPAGE ppsp;
        HPROPSHEETPAGE *phpage; // Pointer auf Seiten-Array
        };
    PFNPROPSHEETCALLBACK pfnCallback;
#if (_WIN32_IE >= 0x0500)
    union {
        HBITMAP hbmWatermark;
        LPCTSTR pszbmWatermark; // Bitmap für linken Rand
        };
    HPALETTE hplWatermark;
    union {
        HBITMAP hbmHeader;
        LPCSTR pszbmHeader;             // Bitmap für Header
        };
#endif
}
```

6.4.5. Die Dialogprozeduren des Assistenten

Hier kann ich mich kurz fassen, da sich am grundlegenden Aufbau der Dialogprozeduren ja nichts ändert. Es kommen lediglich spezielle Benachrichtigungen dazu, die mit dem Assistenten bzw. Property-Sheets allgemein zu tun haben. Bitte lesen Sie darum ggf. den Beitrag über Dialoge.

Neu ist, dass wir die Nachricht "`WM_NOTIFY`" bearbeiten müssen, wenn wir auf Benachrichtigungen des Assistenten reagieren, bzw. diese auch auslösen wollen.

Wert	Bedeutung
`PSN_SETACTIVE`	Die Seite wird angezeigt
`PSN_WIZNEXT`	Der "Weiter"-Button wurde geklickt
`PSN_WIZBACK`	Der "Zurück"-Button wurde geklickt
`PSN_WIZFINISH`	Der "Fertig stellen"-Button wurde geklickt
`PSN_HELP`	Der Hilfe-Button wurde geklickt

(weitere Benachrichtigungscodes finden Sie im PSDK)

Wichtig ist hier, dass die Seiten des Assistenten üblicherweise erst beim Aufruf erzeugt werden. Und **nur dann** können Sie Dialog-typisch "`WM_INITDIALOG`" bearbeiten. Bei jedem weiteren Aufruf der Seite, ob nun über den Weiter- oder Zurück-Button, wird die o.g. Benachrichtigung `PSN_SETACTIVE` ausgelöst.
Sie sollten diese Benachrichtigung in jeder Dialogprozedur bearbeiten, um die notwendigen Buttons zu aktivieren bzw. zu deaktivieren. Nutzen Sie dazu entweder die Nachricht "`PSM_SETWIZBUTTONS`" oder das Makro "`PropSheet_SetWizButtons`". Wenn Sie die Nachricht verwenden, dann übergeben Sie die gewünschten Buttons als `lParam`. Beim Makro geben Sie die Buttons als zweiten Parameter an. Wenn Sie mehrere Buttons aktivieren wollen, kombinieren Sie sie bitte mit **or**. Auf der Begrüßungsseite sollte bspw. nur der Weiter-Button aktiv sein:

```
WM_NOTIFY:
  case PNMHdr(lp)^.code of
    PSN_SETACTIVE:
      PropSheet_SetWizButtons(GetParent(hwndDlg),PSWIZB_NEXT);
  end;
```

Auf den Abbrechen-Button haben Sie keinen Zugriff, der gehört dem zugrunde liegenden Container.

Wie Sie sehen können wurde im Codeauszug "`GetParent(hwndDlg)`" benutzt. Warum das so ist, habe ich im Prinzip bereits gesagt. Das Handle "`hwndDlg`" gehört zu der von Ihnen erstellten Dialogressource. Die Buttons sind allerdings Teil des Assistenten, und weil Ihr Dialog ja Teil des Assistenten ist müssen Sie mit "`GetParent`" auch auf dessen Handle zugreifen.

Auf der letzten Seite ist die Bearbeitung der o.g. Benachrichtigung noch aus einem anderen Grund wichtig: Wenn Sie es vergessen, zeigt Ihre letzte Seite einen Weiter-Button an. Darum müssen Sie hier speziell den "Fertig stellen"-Button (`PSWIZB_FINISH`) angeben, damit sich die Beschriftung und der Typ ändern:

```
    PSN_SETACTIVE:
      PropSheet_SetWizButtons(GetParent(hwndDlg),
        PSWIZB_BACK or PSWIZB_FINISH);
```

Alternativ dazu lässt sich der Button aber auch mit einem anderen Text belegen. Wenn Sie bspw. ein einfaches Setup-Programm schreiben und einen Installieren-Button anbieten wollen, dann tun Sie das am besten beim Bearbeiten von "`WM_INITDIALOG`"

```
WM_INITDIALOG:
  PropSheet_SetFinishText(GetParent(hwndDlg),'Installieren');
```

Dadurch verschwinden aber auch die Zurück- und Weiter-Buttons.

Assistentenseiten überspringen

Um eine Seite zu überspringen, müssen Sie mit "`SetWindowLong`" das Rückgabeergebnis der Dialogprozedur verändern. Dazu übergeben Sie `DWL_MSGRESULT` und den Wert des gewünschten Dialogs, den Sie als nächstes sehen wollen.
Um beispielsweise direkt von der Start- zur Endseite zu gelangen, könnten Sie beim Klick auf den Weiter-Button folgendes tun:

```
WM_NOTIFY:
  case PNMHdr(lp)^.code of
    PSN_WIZNEXT:
      begin
        SetWindowLong(hwndDlg,DWL_MSGRESULT,IDD_FINISHDLG);
```

Allerdings müssen Sie die Nachricht als "bearbeitet" deklarieren, sonst wird der Befehl ignoriert, und Sie landen auf der regulären zweiten Seite:

```
        Result := true;
      end;
```

Im Beispielprogramm gibt es keine Verwendung für so ein Verhalten. Dort ist es aus verschiedenen Gründen erforderlich, die Reihenfolge einzuhalten.

6.4.6. Kurze Übersicht über Property-Sheets

Da Assistenten mit den Property-Sheets vergleichbar sind und auch die selben Records benutzen (von anderen Stilattributen mal abgesehen), wäre dieser Beitrag unvollständig, wenn wir überhaupt nicht darauf eingehen würden.
Ich werde mich auch hier nur auf das Wesentliche beschränken, das beim Erstellen von Property-Sheets interessant ist, und verweise Sie darum auf das Beispiel für die Systemsteuerung, wo ich einen Dialog mit zwei Registerseiten als CPL-Modul benutze.

1. Auch hier müssen Sie für jede gewünschte Registerseite einen eigenen Dialog erstellen. Der Titel wird übrigens als Beschriftung für das Register (landläufig auch Reiter genannt) verwendet.
2. Die Dialogressourcen müssen natürlich auch geladen werden, damit sie dem Programm zur Verfügung stehen. Bei einem Property-Sheet entscheidet die Reihenfolge, in der Sie die Dialoge laden, darüber, wie die Registerseiten angeordnet werden.
   Beachten Sie bitte, dass Ihnen hier zum Teil andere Attribute als bei Assistenten zur Verfügung stehen. So können Sie beispielsweise die Registerseite mit einem Icon ausstatten. Dazu benutzen Sie das Flag `PSP_USEICONID` und geben dann in der `pszIcon`-Membervariablen den Namen der Symbolressource an:

   ```
   psp.dwFlags := PSP_DEFAULT or PSP_USEICONID;
   psp.pszIcon := MAKEINTRESOURCE(1);
   ```

3. Und natürlich müssen Sie die Property-Sheets durch den Befehl "PropertySheet" anzeigen lassen. Auch hierbei sind spezielle Attribute zu beachten. So gibt es bei Property-Sheets natürlich nicht den Header oder die Bitmaps des Assistenten. Dafür wird standardmäßig ein Dialog mit den Buttons OK, Abbrechen und Übernehmen (deaktiviert) erzeugt. Mögliche Attribute sind u.a.

Wert	Bedeutung
PSH_NOAPPLYNOW	versteckt den Übernehmen-Button
PSH_NOCONTEXTHELP	versteckt den Hilfe-Button in der Titelleiste
PSH_PROPTITLE	benötigt die Membervariable `pszCaption` mit einem gültigen Text, der mit dem Präfix "Eigenschaften von" versehen wird
PSH_USEICONID	zeigt ein Symbol in der Titelleiste an (dazu muss `pszIcon` auf eine gültige Ressource verweisen)

6.4.6.1. Besonderheiten in der Dialogprozedur

Die gute Nachricht zuerst: die Dialogprozedur ändert sich eigentlich überhaupt nicht. Neu ist lediglich, dass Sie den Übernehmen-Button im Dialog haben, den Sie ggf. abfragen müssen. Üblicherweise ist dies bei diversen Änderungen in den Registerseiten der Fall. Sie kennen das ja von zahlreichen Dialogen des Systems.
Nun ist der Button aber standardmäßig deaktiviert. Sie müssen dem System daher selbst mitteilen, welche Änderungen so "schwerwiegend" sind, dass dafür der Button aktiviert werden soll. Dies geschieht mit Hilfe der Nachricht "`PSM_CHANGED`", die als `wParam` das Handle des Dialogs benötigt. In der kleinen Demonstration können Sie den Übernehmen-Button aktivieren, indem Sie in irgendwelchen Text in die Eingabezeile auf der ersten Registerseite eintippen:

```
WM_COMMAND:
 case HIWORD(wp) of
   EN_CHANGE:
     SendMessage(GetParent(hwndDlg),PSM_CHANGED,WPARAM(hwndDlg),0);
  end;
```

Wenn der Anwender den Button anklickt, wird die Benachrichtigung "`PSN_APPLY`" als Teil von "`WM_NOTIFY`" ausgelöst. Hier können Sie dann den notwendigen Code ausführen, um alle gemachten Änderungen (je nach Zweck Ihrer Anwendung) zu speichern.
Danach sollten Sie den Button aber wieder deaktivieren, wofür Sie die Nachricht "`PSM_UNCHANGED`" benutzen (wieder mit dem Handle des Dialogs):

```
SendMessage(GetParent(hwndDlg),PSM_UNCHANGED,WPARAM(hwndDlg),0);
```

Manchmal gibt es aber Situationen, in denen nach dem Klick auf Übernehmen das Beenden des Dialogs erforderlich ist. Das kann der Fall sein, wenn Sie gemachten Änderungen so tiefgreifend waren, dass ein normales Abbrechen nicht mehr ausreicht (etwa, wenn ein Systemneustart oder dergleichen notwendig ist). In diesem Fall können Sie mit der Nachricht "`PSM_CANCELTOCLOSE`" bzw. dem Makro "`PropSheet_CancelToClose`" den Abbrechen-Button deaktivieren, wobei sich auch der Text des OK-Buttons in "Schließen" ändert. Als Parameter geben Sie in beiden Fällen nur das Handle zum Property-Sheet an (also zum übergeordneten Fenster Ihrer Dialogressource). Die Werte von `wParam` und `lParam` (bei Verwendung der Nachricht) werden nicht genutzt und bleiben Null.

```
PropSheet_CancelToClose(GetParent(hwndDlg));
```

6.5. Splitter

6.5.1. Den Splitter erzeugen

Bei einem Splitter handelt es sich um kein Control in dem Sinn, wie Sie dies von einem Fenster, Button o.ä. kennen. Der Splitter wird einfach nur "simuliert", indem Sie zwischen den Controls Platz frei lassen. Im Bild erkennen Sie diesen freien Bereich an der Schraffur zwischen Tree-View (links) und List-View (rechts)

Wie Sie auch im Beispielprogramm sehen können, werden die drei Controls (Tree-View, List-View und Edit-Control) jeweils an der gleichen Position im Fenster erzeugt, und das Edit-Control hat zu diesem Zeitpunkt nicht einmal Breite und Höhe:

```
hTreeView  := CreateWindowEx(WS_EX_CLIENTEDGE,WC_TREEVIEW,nil,
  WS_VISIBLE or WS_CHILD or TVS_HASLINES or TVS_LINESATROOT
  or TVS_HASBUTTONS or TVS_EDITLABELS,0,0,10,10,wnd,
  IDC_TREEVIEW,hInstance,nil);

hLV        := CreateWindowEx(WS_EX_CLIENTEDGE,WC_LISTVIEW,nil,
  WS_VISIBLE or WS_CHILD or LVS_ICON or LVS_AUTOARRANGE or
  LVS_SHAREIMAGELISTS,0,0,10,10,wnd,IDC_LV,hInstance,nil);

hMemo      := CreateWindowEx(WS_EX_CLIENTEDGE,'Edit',nil,
  WS_CHILD or WS_VISIBLE or WS_VSCROLL or ES_MULTILINE or
  ES_NOHIDESEL,0,0,0,0,wnd,IDC_MEMO,hInstance,nil);
```

Aber das spielt keine Rolle, denn die Neupositionierung und die Änderung der Größe findet in "`WM_SIZE`" statt.

Zunächst sind aber ein paar Konstanten und Variablen erforderlich, mit denen wir uns die Arbeit ein wenig erleichtern wollen. Der Einfachheit halber habe ich die minimale Breite und Höhe sowie die Breite des Splitters an sich als Konstanten deklariert.
Die `integer`-Variablen dienen später zur Ermittlung der Position, und die beiden `bool`-Variablen werden als Flags benutzt, damit das Programm "weiß", wann ein Splitter gezogen wird und wann nicht

```
const
  MINVSPLIT     =   190;
  MINHSPLIT     =   100;
  SPLITWIDTH    =     3;
var
  vSplitPos     : integer = MINVSPLIT + 30;
  hSplitPos     : integer = wHeight - (MINHSPLIT + 50);
  fVTrackSplit  : boolean = false;
  fHTrackSplit  : boolean = false;
```

Wie gesagt wird nun in "`WM_SIZE`" dafür gesorgt, dass a) die Controls entsprechend neu positioniert werden, und dass sie b) auch die neuen Größen erhalten, damit die beiden Splitter quasi sichtbar werden.
Sollten allerdings die Maße des Fensters so gering sein, dass die Splitter nicht mehr zu sein sein würden, wird ihre Position ggf. zurückgesetzt. Beachten Sie bitte, dass die beiden Variablen "`i`" und "`rc`" im Beispielprogramm vorher ermittelt werden und die Höhe der Statuszeile und die Maße des Clientbereichs des Fensters enthalten

```
if(vSplitPos >= rc.Right - MINVSPLIT) then
  vSplitPos := rc.Right - MINVSPLIT;
if(hSplitPos >= rc.Bottom - i - MINHSPLIT) then
  hSplitPos := rc.Bottom - i - MINHSPLIT;
```

Nun können wir uns an die Controls heranwagen:

Zuerst verschieben wir den Tree-View auf seine neue Position. Da er links sein soll, sind zumindest die X- und Y-Werte kein Problem und richten sich nach den Angaben im Rechteck "`rc`". Für die Breite verwenden wir die Variable "`vSplitPos`", und für die Höhe "`hSplitPos`"

```
MoveWindow(hTreeView,rc.Left,rc.Top,vSplitPos,hSplitPos,true);
```

Die List-View hat mit dem Tree-View zumindest die Y-Koordinate und die Höhe gemeinsam: nämlich den oberen Rand des Clientbereichs des Fensters sowie die Variable "`hSplitPos`". Der X-Wert allerdings richtet sich ebenfalls nach der Variablen "`vSplitPos`", wobei hier aber noch die Breite des Splitters berücksichtigt werden muss. Schließlich wollen Sie ja den gewünschten Platz zwischen den beiden Controls lassen. Und die Breite des Controls errechnet sich aus der Angabe in der schon erwähnten `TRect`-Variablen, wobei hier natürlich die Splitterposition und -breite abgezogen werden müssen

```
MoveWindow(hLV,vSplitPos + SPLITWIDTH,rc.Top,rc.Right -
  (vSplitPos + SPLITWIDTH),hSplitPos,true);
```

Damit bleibt noch das mehrzeilige Editfeld (quasi ein Memo), das im unteren Teil des Fensters sitzt und durch einen horizontalen Splitter von Tree-View und List-View getrennt ist.
Hier sind die Angaben für den linken und rechten Rand recht einfach, weil der Splitter ja über die gesamte Fensterbreite gehen soll. Als Höhe wird dagegen die Variable "`hSplitPos`" (mit der Angabe zur Splitterposition) herangezogen, wobei auch hier die Konstante mit dem Breiten- bzw. (in dem Fall) Höhenwert addiert werden muss, damit Sie den Splitter auch sehen. Und die Höhe des Editfeldes selbst richtet sich nach dem unteren Rand des Fensters, von dem natürlich die Höhe der Statuszeile und die Splitterposition und -breite abgezogen werden müssen

```
MoveWindow(hMemo,rc.Left,hSplitPos + SPLITWIDTH,rc.Right,
  rc.Bottom - i - (hSplitPos + SPLITWIDTH),true);
```

Damit wäre der erste Schritt vollbracht. Die "Splitter" reagieren zwar bisher auf überhaupt nichts, aber es hat zumindest schon einmal den Anschein, als wäre da was ... :o)

6.5.2. Den Splitter kenntlich machen

Zuerst wollen wir dafür sorgen, dass die Splitter auch visuell als solche erkannt werden. Dazu ist es einfach nur notwendig, einen speziellen Mauszeiger zu benutzen, sobald die Maus in den Bereich des Splitters kommt.
Und das geht am einfachsten bei der Bearbeitung der Nachricht "`WM_MOUSEMOVE`", die im `lParam` die aktuellen Koordinaten liefert. Das niederwertige `word` enthält dabei die X-Koordinate, das höherwertige die Y-Koordinate:

```
WM_MOUSEMOVE:
  begin
    x := LOWORD(lp);
    y := HIWORD(lp);

    if(x >= vSplitPos) and (x <= vSplitPos + SPLITWIDTH) then
      SetCursor(LoadCursor(0,IDC_SIZEWE))
    else if(y >= hSplitPos) and (y <= hSplitPos + SPLITWIDTH) then
      SetCursor(LoadCursor(0,IDC_SIZENS));
  end;
```

Die beiden benutzen Konstanten `IDC_SIZEWE` und `IDC_SIZENS` entsprechen dem Mauszeiger, der üblicherweise für Splitter verwendet wird: Er zeigt einen Doppelpfeil in Richtung West-Ost (`ISC_SIZEWE`) bzw. Nord-Süd (`ISC_SIZENS`) an, bspw.:

6.5.3. Den Splitter nutzen

Um den Splitter nun verwenden zu können, müssen Sie kurz die Funktionsweise überdenken:

1. Die linke Maustaste wird gedrückt und muss gedrückt gehalten werden.
2. Bei gedrückt gehaltener Maustaste wird die Maus in die gewünschte Richtung verschoben, wobei die aktuelle Splitterposition meist durch eine Art Schatten kenntlich gemacht wird.
3. Ist die neue Position erreicht, wird die Maustaste losgelassen, und die von der Änderung betroffenen Controls werden entsprechend angepasst.

Damit wäre der Weg eigentlich schon vorgegeben -

Zuerst reagieren Sie auf den Druck der linken Maustaste. Die entsprechende Nachricht heißt "`WM_LBUTTONDOWN`", und sie liefert die aktuellen Koordinaten im `lParam` zurück. Die Vorgehensweise unterscheidet sich für den vertikalen und horizontalen Splitter nur in Details, der Code ist so gesehen aber identisch. Darum hier die Erklärung für den vertikalen Splitter, der Tree-View und List-View trennt.
Wie gesagt, Sie benötigen zuerst einmal die aktuelle Position des Mauszeigers. Beim vertikalen Splitter ist dabei die X-Koordinate im niederwertigen `word` wichtig:

```
x := LOWORD(lp);
```

Wenn die Position des Mauszeigers innerhalb der Splitterposition liegt,

```
if(x >= vSplitPos) and (x <= vSplitPos + SPLITWIDTH) then
begin
```

dann setzen Sie den entsprechenden Cursor und sorgen mit "`SetCapture`" dafür, dass die Mausnachrichten nur an das eigene Fenster geleitet werden.

```
  SetCursor(LoadCursor(0,IDC_SIZEWE));
  SetCapture(wnd);
```

Diese Funktion wurde übrigens auch schon beim Drag&Drop des Tree-View angesprochen.

Der nächste Schritt ist der oben erwähnte "Schatten". Ich möchte es auch weiterhin so bezeichnen. Gemeint ist damit natürlich die Schraffur, die Sie mit der Maus verschieben, und die die dann neue Position des Splitters kennzeichnet. Für diesen Schatten habe ich die Funktion "`DrawTrackSplit`" von Eugen 1:1 übernommen

```
  DrawTrackSplit(wnd,vSplitPos,0,SPLITWIDTH,hSplitPos);
```

Außerdem ist wichtig, dass Sie die bool'sche Variable für den vertikalen Splitter setzen. Das ist das interne Signal, dass das Verschieben der Controls begonnen hat:

```
  fVTrackSplit := true;
end
```

Der nächste Schritt führt uns zurück zur "`WM_MOUSEMOVE`"-Nachricht, denn natürlich muss die Maus bewegt werden um dem Splitter eine neue Position zu geben. Im vorigen Kapitel wurde gezeigt, wie man diese Nachricht zur Kennzeichnung des Splitters nutzt. An der Stelle genügt eine Erweiterung für den vertikalen oder horizontalen Splitter, der sich nach dem Status der dazu gehörenden `bool`-Variablen richtet.

Wie ich bereits sagte, wird beim Druck auf die linke Maustaste ermittelt, ob die aktuellen Koordinaten der Maus zu einem

der Splitter passen. Das obige Beispiel demonstrierte Ihnen das für den vertikalen Splitter. Also bleiben wir auch dabei und gehen davon aus, dass dieser Splitter benutzt werden soll.
In diesem Fall ist die Variable "`fVTrackSplit`" **true**, wovon wir die folgenden Codezeilen abhängig machen:

```
if(fVTrackSplit) then
begin
```

Zuerst stellen wir sicher, dass der Splitter nicht zu weit bewegt werden kann. Das bringt mich wieder auf die Konstanten, die ich eingangs deklariert habe. Wenn Sie also die Grenze des jeweiligen Splitters unterschreiten, bleibt der Schatten des Splitters auf der letztmöglichen Position stehen:

```
  if(x < MINVSPLIT) then x := MINVSPLIT
    else if(x > rc.Right - MINVSPLIT) then x := rc.Right - MINVSPLIT;
```

Eugen hat an dieser Stelle in seiner Demo einen recht witzigen Snap-Effekt, d.h. wenn Sie einen bestimmten Wert unterschreiten, springt der Splitter regelrecht an den jeweiligen Rand. Diesen Effekt können Sie sehen, wenn Sie in der Zeile

```
{.$DEFINE SNAPFX}
```

den Punkt entfernen und so den Compilerschalter aktivieren. Die technische Umsetzung dieses Effekts sollte eigentlich kein Problem sein: Sie prüfen lediglich ob ein bestimmter Schwellenwert erreicht bzw. unterschritten wurde und setzen die jeweilige Koordinate dann einfach auf den kleinsten möglichen Wert, etwa

```
  if(x < rc.Left + 50) then x := rc.Left + SPLITWIDTH
    else if(x > rc.Right - 50) then x := rc.Right - SPLITWIDTH;
```

Wenn Sie nun bemerken, dass die Position des Splitters (in dem Fall "`vSplitPos`") nicht mehr mit der jeweils ermittelten Koordinate übereinstimmt

```
  if(vSplitPos + 1 <> x) then
  begin
```

dann entfernen Sie zuerst den Schatten des Splitters an der aktuellen Position. Das lässt sich einfach dadurch erreichen, dass Sie die schon erwähnte Funktion "`DrawTrackSplit`" erneut aufrufen. Die beiden so erzeugten Grafiken (das steckt eigentlich dahinter) heben sich dadurch gegenseitig auf

```
    DrawTrackSplit(wnd,vSplitPos,0,SPLITWIDTH,hSplitPos);
```

Dann passen Sie die Position an und zeichnen den Schatten an der neuen Position

```
    inc(vSplitPos,x - vSplitPos - 1);
    DrawTrackSplit(wnd,vSplitPos,0,SPLITWIDTH,hSplitPos);
  end;
end
```

Der nächste Schritt: Sie haben die gewünschte Position erreicht und lassen die Maustaste wieder los. In dem Fall rufen wir einfach nur "`ReleaseCapture`", damit die Mausnachrichten wieder normal verarbeitet werden und nicht mehr auf unser Fenster beschränkt sind. Die dafür zu bearbeitende Nachricht heißt "`WM_LBUTTONUP`"

```
WM_LBUTTONUP:
  if fVTrackSplit or fHTrackSplit then
    ReleaseCapture;
```

Und das bringt uns direkt zum letzten Schritt: Wenn wir "`ReleaseCapture`" aufrufen, dann sendet das System die Nachricht "`WM_CAPTURECHANGED`", die wir uns zunutze machen. In unserem Beispiel ist nach wie vor der vertikale Splitter aktiv, erkennbar am Status **true** der Variablen "`fVTrackSplit`". Wenn dies der Fall ist, dann entfernen wir zunächst den Schatten des Splitters

```
if(fVTrackSplit) then
begin
  DrawTrackSplit(wnd,vSplitPos,0,SPLITWIDTH,hSplitPos);
```

und senden dann einfach "WM_SIZE" an unsere Anwendung. Dadurch lösen wir das Neuzeichnen bzw. das Neuanordnen der Controls aus, das anfangs bereits angesprochen wurde

```
  SendMessage(wnd,WM_SIZE,0,0);
```

Und natürlich muss die bool-Variable auch wieder zurückgesetzt werden, weil die Arbeit mit dem Splitter an der Stelle ja beendet ist

```
  fVTrackSplit:= false;
end;
```

6.5.4. Der Splitter-Schatten

Zum Abschluss soll noch Eugens Funktion "DrawTrackSplit" genauer unter die Lupe genommen werden, da sie den "Schatten" des Splitters erzeugt.

```
procedure DrawTrackSplit(const hWnd: HWND; x, y, Width, Height: longint;
  bPattern: boolean = true);
const
  DotBits : array[0..7] of Word =
    ($5555,$AAAA,$5555,$AAAA,$5555,$AAAA,$5555,$AAAA);
var
  dc      : HDC;
  hbr     : HBRUSH;
  bmp     : HBITMAP;
begin
```

Zuerst benötigen wir ein Handle auf den *Device Context* des Fensters

```
  dc      := GetDCEx(hWnd,0,DCX_CACHE or DCX_CLIPSIBLINGS or
    DCX_LOCKWINDOWUPDATE);
```

Da der Parameter "bPattern" standardmäßig auf **true** steht, wird die schon erwähnte Schraffur erzeugt. Diese Schraffur ist eine im Speicher erzeugte Bitmap, mit einem Raster, das sich aus dem Array der Konstante "DotBits" zusammensetzt. Würden Sie "bPattern" auf **false** setzen, würden Sie statt der Schraffur einen schwarzen Strich sehen

```
  if bPattern then
  begin
    bmp := CreateBitmap(8, 8, 1, 1, @DotBits);
    hbr := SelectObject(dc, CreatePatternBrush(bmp));
    DeleteObject(bmp);
  end;
```

Mit "PatBlt" wird das Muster nun auf das Fenster gezeichnet. Durch die Verwendung von PATINVERT wird das Muster bei einem nochmaligen Aufruf wieder entfernt.

```
  PatBlt(dc, x, y, Width, Height, PATINVERT);
```

Die im Speicher erzeugte Bitmap und das Handle des *Device Context* müssen noch freigegeben werden

```
  if bPattern then
    DeleteObject(SelectObject(dc, hbr));

  ReleaseDC(hWnd, dc);
end;
```

und das war´s.

7. Hilfedateien erstellen und nutzen

7.1. HLP-Hilfedateien

7.1.1. Vorwort

Viele Entwickler schreiben, so wie ich, in erster Linie einfach nur ihre Programme und betrachten die Angelegenheit damit als erledigt. Spätestens wenn sich die Frage nach einer Veröffentlichung stellt, wird auch das Thema einer Dokumentation interessant. Nun könnte man natürlich eine Textdatei beilegen, die die grundlegenden Funktionen der Software erklärt. Aber weitaus professioneller wirkt eine Hilfedatei.

Hilfedateien im HLP-Format sind seit Windows 3.x bekannt, auch wenn sie sich seit Windows 95 ein wenig verändert haben. Das Schöne dabei ist, dass Delphi diese Hilfedateien direkt unterstützt, so dass aus programmiertechnischer Sicht nichts weiter benötigt wird.

Auf redaktioneller Ebene benötigen wir ein Schreibprogramm, das RTF-Dateien erzeugen kann. Außerdem sollte es in der Lage sein, unsichtbaren Text darzustellen (das hört sich paradox an, hat aber durchaus einen ernsthaften Hintergrund). Des Weiteren sollten einfache und doppelte Unterstreichungen, Fußnoten usw. kein Problem darstellen.
Die Grafiken dieses Tutorials basieren auf Microsoft Word XP. Aber das zugrunde liegende Prinzip funktioniert natürlich auch mit anderen Programmen, die die genannten Anforderungen erfüllen.

Es gibt auch kommerzielle Software sowie Freeware, die die Erstellung von HLP-Dateien erleichtert. Ich muss ehrlicherweise zugeben, dass ich mich mit diesen Produkten nicht auskenne. Wer dieses Tutorial dahingehend ergänzen möchte, kann mir gern schreiben.

Für unsere Zwecke benötigen wir zu guter Letzt noch den kostenlosen Helpworkshop von Microsoft, der sowohl bei Delphi als auch beim VisualStudio 6 mitgeliefert wird.

Tipp

Für Anfänger, die wirklich in die HLP-Materie einsteigen wollen, gibt es ein recht gutes Tutorial, die 99 Steps to WinHelp von Hans-Jürgen Philippi. Aber auch für Fortgeschrittene findet sich hier der ein oder andere nützliche Tipp. Allerdings fehlt in meinen Augen das Inhaltsverzeichnis. Zwar sind die 99 Steps eine Art A-Z-Tutorial, bei dem man "gezwungen" wird, sich vom Anfang bis zum Ende durchzuarbeiten. Wenn man aber die Tipps ausprobiert hat und vielleicht einen Fehler korrigieren muss, den man erst im Probelauf bemerkt hat, dann steht man vor dem Problem, erst bis zum entsprechenden Thema blättern zu müssen - wenn man kein Lesezeichen angelegt hat.

Es wäre daher wünschenswert, dass Hans-Jürgen Philippi seinem Werk eine kleine Aktualisierung spendiert. Ein Inhaltsverzeichnis ließe sich - um die Ausrichtung des Dokuments für Anfänger nicht zu ändern - ja auch als Download auf der letzten Hilfeseite anbieten.

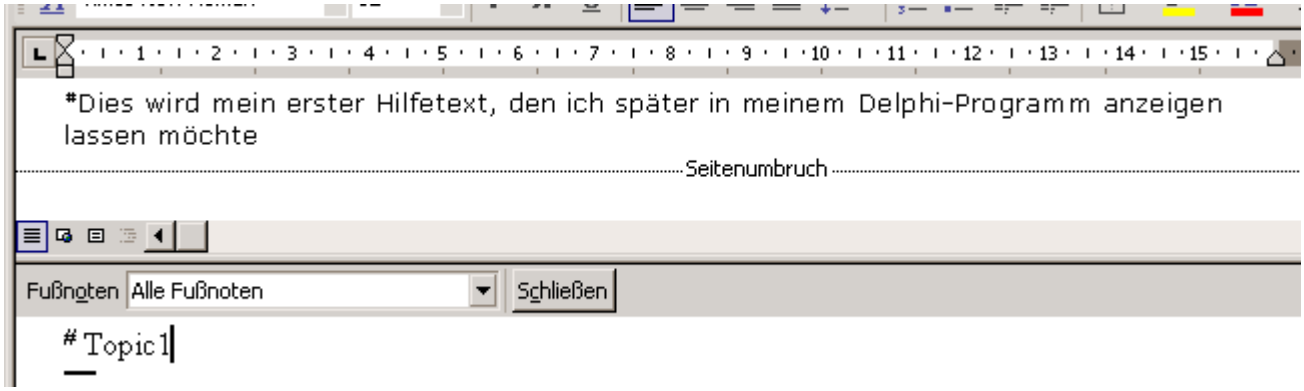
7.1.2. Der erste Hilfetext

Wir starten also unser Schreibprogramm und beginnen mit dem ersten Hilfetext. Das kann ein Einzeiler sein, ein mehrzeiliger Absatz ... mehrere Seiten ... ganz wie Sie mögen. Selbstverständlich können auch Grafiken eingefügt werden. Laut einer alten PC PRAXIS sind auch Tabellen möglich, allerdings ohne Rahmen. Ich muss zugeben, ich habe das nicht probiert.

> *Hinweis*
>
> Zu beachten ist, dass Hintergrundfarben in der RTF-Datei ignoriert, bzw. durch die Farbeinstellungen des Fensters ersetzt werden. Nur eine geänderte Schriftfarbe wird akzeptiert und angezeigt. Sie sollten daher schon im Vorfeld die Hintergrundfarbe des Hilfebetrachters berücksichtigen, damit es nicht zu ungünstigen Farbkombinationen kommt.

Um die einzelnen Hilfethemen voneinander zu trennen, muss ein manueller Seitenumbruch eingefügt werden.

Neben der Trennung durch den Seitenumbruch benötigt jedes Thema einen internen Bezeichner, damit später bei der Auswahl des Themas aus dem Inhaltsverzeichnis auch der korrekte Hilfetext zu sehen ist. Dazu verwenden wir benutzerdefinierte Fußnoten, die wir am Beginn unseres Hilfetextes einfügen:



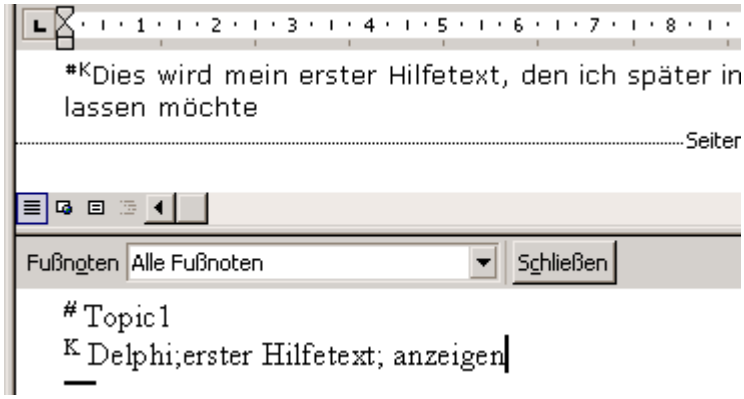
Sie sehen, dass ich als Fußnote das Doppelkreuz **#** benutzt habe. Es stehen mehrere Typen zur Verfügung, die wichtigsten davon sind:

```
#         für die Identifikation
$         für Überschriften
K         für Schlüsselworte im Index
+         für Browse-Sequenzen
```

Verwenden Sie möglichst keine Leerzeichen für die Bezeichner der Fußnoten. Nutzen Sie stattdessen z.B. den Unterstrich. Damit vermeiden Sie Probleme beim Zugriff auf die einzelnen Hilfethemen.

7.1.3. Der Index

Für den Index unserer Hilfedatei verwenden wir ebenfalls Fußnoten, wie schon angedeutet. Das Prinzip ist dabei recht einfach: wir fügen die Fußnote **K** hinzu und geben die gewünschten Indexbegriffe an, die wir mit dem Thema verknüpfen wollen:



Wenn Sie mehrere Begriffe nutzen, dann trennen Sie sie durch ein Semikolon voneinander (wie Sie im Bild sehen können).

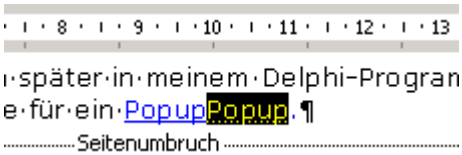
7.1.4. Sprungmarken

Weil auch der Hilfebetrachter von Windows immer nur ein Thema anzeigt, muss es eine Möglichkeit geben, innerhalb des Textes zu einem anderen Thema zu wechseln. Was man im Web landläufig als Link bezeichnet, nennt sich hier "Sprungmarke", wobei wir auch hierfür die internen Namen unserer Fußnote **#** heranziehen. Da es nicht falsch ist, nenne ich die Sprungmarken im folgenden auch einfach Link.

Es gibt zwei Arten: Der normale Link zeigt ein Hilfethema im Fenster an, der Popup-Link dagegen öffnet zusätzlich ein

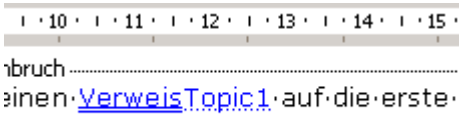
kleines Popup-Fenster mit der entsprechenden Information.

Über die Art entscheidet dabei die Unterstreichung im Text. Eine doppelte Unterstreichung steht für den normalen Link, eine einfache für den Popup-Link. Das Bild zeigt Ihnen ein Beispiel für einen Popup-Link:



Tatsächlich verantwortlich für das Funktionieren eines solchen Links ist allerdings der unsichtbare Text. Er sorgt für die Identifizierung des gesuchten Themas. Zur besseren Bearbeitung sollten Sie Ihr Schreibprogramm so einstellen, dass es Ihnen den unsichtbaren Text anzeigt. Die gepunktete Unterstreichung des hinteren Wortes "Popup" ist nämlich die eigentliche Sprungmarke (besser gesagt: der Verweis auf die Fußnote des gewünschten Themas), hierbei handelt es sich normalerweise um ausgeblendeten Text.
Davor sehen Sie ebenfalls den Begriff "Popup", der einmal unterstrichen ist. Dieser Text wäre später in Ihrer Hilfedatei zu sehen und würde - wenn Sie ihn anklicken - ein Popupfenster mit dem entsprechend zugeordneten Thema öffnen.

Als Beispiel zeige ich Ihnen noch einen normalen Link, der ein neues Hilfethema öffnen würde. Und wie Sie sehen ist tatsächlich nur die Unterstreichung eine andere:



7.1.5. Das Inhaltsverzeichnis

Für das Inhaltsverzeichnis verwenden wir den eingebauten Editor des Helpworkshop. Starten Sie das Programm und wählen Sie im Menü "File/New/Help Contents".

Da meine Beispieldatei drei Hilfethemen hat, erstelle ich auch drei Einträge.

Den allerersten Eintrag fügen wir mit dem Button "Add Above" oder "Add Below" hinzu. Das geht, weil wir bis zu diesem Zeitpunkt noch nichts angegeben haben. Sind schon Einträge im Inhaltsverzeichnis, entscheiden die beiden Buttons, wo der neue Eintrag angeordnet wird.

Zuerst wollen wir eins der typischen Bücher darstellen, mit denen man Themen gruppieren kann. Klicken Sie also auf "Add Above" und stellen Sie als Option "`Heading`" ein. Dadurch werden die meisten Optionen deaktiviert, so dass Sie lediglich unter "Title" einen gewünschten Begriff angeben müssen.

Nun wollen wir aber die eigentlichen Hilfethemen eintragen. Sorgen Sie dafür, dass der erste Eintrag im Editor markiert ist und klicken Sie dann auf "Add Below". Lassen Sie die Option diesmal auf "`Topic`" und geben Sie erst einmal den Titel an, unter dem das Hilfethema im Inhaltsverzeichnis erscheinen soll.
Wenn Sie das gemacht haben, tragen Sie unter "Topic ID" den Namen der Fußnote ein, die dem Thema zugeordnet ist.

Wiederholen Sie das bei Ihrer Hilfedatei mit allen Einträgen, die Sie im Inhaltsverzeichnis zeigen wollen.

Wenn Sie fertig sind, speichern Sie das Inhaltsverzeichnis. Um Verwirrungen zu vermeiden sollten Sie für Hilfedatei und Inhaltsverzeichnis den selben Namen verwenden.

7.1.6. Das Hilfeprojekt beginnen

Starten Sie ggf. den Helpworkshop und beginnen Sie mit Hilfe des Menüs "File/New/Help Project" ein neues Hilfeprojekt, dem Sie zuerst einen Namen geben müssen.

Um unser Projekt mit Leben zu füllen, klicken wir auf den Button "FILES" und wählen zuerst unsere RTF-Datei mit dem Hilfetext aus. Wenn Sie die Texte auf mehrere Dateien aufgeteilt haben, dann wählen Sie bitte alle relevanten Dateien aus.

Wenn Sie ein Inhaltsverzeichnis erstellt haben und nutzen möchten, dann klicken Sie bitte auf den Button "OPTIONS" und wählen Sie auf der Registerseite "Files" unter "Contents file" Ihre Datei mit dem Inhaltsverzeichnis aus.

Wenn Sie wollen, dann lassen Sie die Hilfedatei erstellen, und probieren Sie sie aus!

7.1.7. Ein Zielfenster definieren

Standardmäßig zeigt der Hilfebetrachter von Windows ein festgelegtes Fenster mit ein paar Buttons an. Wir können diese Vorgaben aber auch ändern, indem wir ein eigenes Fenster definieren. Das wäre hilfreich, wenn Sie in Ihrer Hilfedatei z.B. keinen Index verwenden - dann könnten Sie den gleichnamigen Button in der Toolbar des Hilfebetrachters verbergen.

Klicken Sie im Helpworkshop auf den Button "WINDOWS" und geben zunächst einen Namen für das Fenster an, z.B. "`HLPWIN`". Dieser Name wird nicht im Text angezeigt, er dient nur als interner Bezeichner. Wenn Sie das getan haben, haben Sie Zugriff auf verschiedene Optionen:

Auf der Registerseite "Position" können Sie z.B. die Größe des Hilfefensters einstellen.
Auf der Seite "Buttons" können Sie angeben, welche Schaltknöpfe im Hilfebetrachter zu sehen sein sollen.
Unter "Color" können Sie die Farbe des Hilfefensters ändern.

Ich schlage vor, dass Sie ein wenig mit den Optionen experimentieren.

Wichtig ist nur, dass Sie dieses Fenster in Ihrem Inhaltsverzeichnis (sofern vorhanden) definieren, damit Ihre Einstellungen auch tatsächlich wirksam werden. Dazu laden Sie bitte die Datei mit Ihrem Inhaltsverzeichnis und tragen unter "Default filename (and window)" den Namen Ihrer Hilfedatei ein. Hängen Sie dann ein Größer-als-Zeichen an und ergänzen Sie den Namen des Fensters, beispielsweise:

```
Sample.hlp>HLPWIN
```

7.1.8. Die kontextsensitive Hilfe

Immer sehr schön ist diese Form der Hilfe. Das bedeutet, der Anwender kann ein Kontrollelement des Programms auswählen und erhält dann gezielt eine Information über Sinn und Zweck des Elements. Auf den technischen Hintergrund, der uns als Delphi-Programmierer interessiert, gehen wir noch ein.

An dieser Stelle müssen wir unsere Hilfethemen mit numerischen Werten verknüpfen, da Delphi von uns einen Zahlenwert für die Kontexthilfe erwartet. Wir laden also unser Hilfeprojekt und klicken auf den Button "MAP". Mit "Add" fügen wir unsere erste (ich nenne es mal:) Verknüpfung hinzu. Als Beispiel werde ich den Popup-Text meiner Hilfedatei mappen:

```
Topic ID:                   Popup
Mapped numeric value:       1000
```

Nach diesem Muster verknüpfen Sie bitte alle gewünschten Hilfethemen mit numerischen Werten. Der Wert 1000 richtet sich in meinem Fall übrigens nach der Eigenschaft "**HelpContext**", die Sie im Objektinspektor (VCL) finden, wenn Sie das gewünschte Element auswählen. Bei nonVCL-Projekten können Sie diesen Wert mit Hilfe der Funktion "`SetWindowContextHelpId`" wählen. Die tatsächlichen Werte richten sich also immer nach Ihren Angaben. Wenn die

kontextsensitive Hilfe nicht funktioniert, kontrollieren Sie bitte die numerischen IDs.

Speichern Sie das Projekt neu ab, und lassen Sie ggf. auch die Hilfedatei neu erstellen.

7.1.8.1. Hilfeseiten gezielt aufrufen

Auf die gleiche Weise können Sie übrigens auch Hilfethemen mappen, wenn Sie diese später gezielt in Ihrem Programm aufrufen wollen. Für unser Beispiel wollen wir die zweite Hilfeseite herausgreifen:

```
Topic ID:                   Topic2
Mapped numeric value:       2000
```

Außerdem mappen wir noch eine spezielle Seite, die wir später in einer MessageBox anzeigen lassen wollen:

```
Topic ID:                   MessageBox
Mapped numeric value:       3000
```

7.1.9. Die Hilfe verwenden

Der Aufruf der Hilfedatei geht relativ einfach. Zuerst sollten wir natürlich sicherstellen, dass die Hilfedatei überhaupt vorhanden ist. Ist dies nicht der Fall, deaktivieren wir alle Elemente und Funktionen, die irgendwie mit der Hilfe zu tun haben. Im Beispielprogramm sieht das wie folgt aus:

```
if(fileexists(szHelpFile) = FALSE) then
  begin
    // Buttons deaktivieren
    EnableWindow(hTocBtn,false);
    EnableWindow(hIdxBtn,false);
    EnableWindow(hTheme2Btn,false);

    // Hilfe-Button aus der Titelleiste des Fensters
    // entfernen
    SetWindowLong(wnd,GWL_EXSTYLE,
      GetWindowLong(wnd,GWL_EXSTYLE) and not WS_EX_CONTEXTHELP);
  end;
```

Wenn wir sicher sind, dass unsere Hilfedatei auch vorhanden ist, dann rufen wir den API-Befehl "`WinHelp`" auf und lassen uns z.B. das Inhaltsverzeichnis anzeigen:

```
if(IsWindowEnabled(hTocBtn)) then
  WinHelp(wnd,@szHelpFile[1],HELP_FINDER,0);
```

Um gezielt eine bestimmte Hilfeseite aufzurufen, müssen Sie diese gemappt haben, da die Windows-Hilfe mit numerischen Werten arbeitet. In unserem Beispiel haben wir der zweiten Hilfeseite die ID 2000 zugeordnet, so dass wir diese Seite jetzt anzeigen können:

```
if(IsWindowEnabled(hTheme2Btn)) then
  begin
    URL := szHelpFile + '>HLPWIN';
    WinHelp(wnd,@URL[1],HELP_CONTEXT,2000);
  end;
```

Hinweis

Das Beispiel zeigt Ihnen u.a. auch, wie man die Ausgabe eines Hilfethemas in ein definiertes Fenster umleitet. Wenn Sie kein Fenster definiert haben, benutzen Sie stattdessen nur den Namen der Hilfedatei zur Übergabe an den API-Befehl "`WinHelp`".

WinHelp-Definition

```
BOOL WinHelp(
    HWND hWndMain,           // Handle des Parent-Fensters
    LPCTSTR lpszHelp,        // Name der Hilfedatei
    UINT uCommand,           // Hilfe-Kommando
    DWORD dwData             // Datenwert
);
```

7.1.9.1. F1 benutzen

Die Taste F1 wird üblicherweise zur Anzeige der Hilfedatei eines Programms verwendet. Allerdings (und dazu muss ich leider ein wenig auf das nächste Kapitel vorgreifen) dient F1 auch dazu, Kontextinformationen zu einem aktiven Control anzuzeigen. Wir müssen also unterscheiden, wann welche Hilfe erwünscht ist.
Dabei hilft uns das `THelpInfo`-Record, auf das wir über den Zeiger im `lParam` zugreifen können, wenn wir die Nachricht "`WM_HELP`" auswerten. Da uns hier nur die Anzeige der Hilfedatei interessiert (die kontextsensitive Hilfe folgt im nächsten Kapitel, wie gesagt!), prüfen wir, ob das im Record übermittelte Handle mit dem Fensterhandle identisch ist. Ist das der Fall, dann senden wir lediglich einen Buttonklick mit der ID des Hilfe-Buttons an das Programm. Dadurch sparen wir auch gleich noch Code:

```
WM_HELP:
  if(PHelpInfo(lp)^.iContextType = HELPINFO_WINDOW) then begin
    if(PHelpInfo(lp)^.hItemHandle = wnd) then
      SendMessage(wnd,WM_COMMAND,MAKELONG(IDC_TOCBTN,BN_CLICKED),0)
  end else
    Result := DefWindowProc(wnd,uMsg,wp,lp);
```

7.1.9.2. Die Hilfe in einer MessageBox verwenden

Hier gibt es eigentlich zwei Möglichkeiten. Die erste ist die Benutzung der Befehle "`MessageBox`" bzw. "`MessageBoxEx`". Um die Hilfe hier verwenden zu können, benutzen Sie das Flag `MB_HELP` bei der Angabe des Stils, beispielsweise

```
MessageBox(wnd,'Klicken Sie auf den Hilfe-Button',
  AppName,
  MB_OKCANCEL or MB_HELP);
```

Wenn der Anwender diesen Button anklickt oder F1 drückt, wird die Nachricht "`WM_HELP`" an das Besitzerfenster (unsere Anwendung) geschickt. Und damit ergibt sich ein kleines Problem. Wie Sie etwas weiter oben sehen können, prüfen wir in "`WM_HELP`" bisher nur, ob das übermittelte Handle dem Handle der Anwendung entspricht. Diesmal ist es aber das Handle der Dialogbox, und deshalb müssen wir die Funktion noch etwas erweitern:

```
if(PHelpInfo(lp)^.hItemHandle = wnd) or
  (GetParent(PHelpInfo(lp)^.hItemHandle) = wnd)
then
  SendMessage(wnd,WM_COMMAND,MAKELONG(IDC_TOCBTN,BN_CLICKED),0)
```

Voilà.

Ich möchte aber noch auf eine etwas elegantere Methode hinweisen, bei der die Dialogbox auch ein ganz bestimmtes Thema der Hilfe anzeigen kann. Dazu benötigen wir aber "`MessageBoxIndirect`", die die Dialogbox mit Hilfe eines `TMsgBoxParams`-Records erzeugt:

```
const
  HLP_PAGE_4     = 3000;
var
  msgbox         : TMsgBoxParams = (
    cbSize:sizeof(TMsgBoxParams);
    lpszText:'Klicken Sie auf den Hilfe-Button';
    lpszCaption:AppName;
    dwStyle:MB_OKCANCEL or MB_HELP or MB_ICONINFORMATION;
    dwContextHelpId:HLP_PAGE_4;
  );
```

Sie sehen hier die Membervariable `dwContextHelpId`, die zur Anzeige eines ausgewählten Themas der Hilfedatei benutzt wird. In diesem Fall ist das eine Seite, die den gemappten Wert 3000 besitzt. Um die Dialogbox dann aufzurufen, setzen wir zunächst noch die Werte des Besitzerfensters und des Instanzenhandles, dann benutzen wir o.g. Funktion und übergeben als Parameter das `TmsgBoxParams`-Record:

```
msgbox.hwndOwner           := wnd;
msgbox.hInstance           := hInstance;
MessageBoxIndirect(msgbox);
```

Wenn Sie die Dialogbox nun anzeigen lassen und auf den Hilfe-Button klicken, dann erscheint der Text des gemappten Themas als Popup. Das liegt daran, dass das Beispielprogramm natürlich schon die Behandlung der kontextsensitiven Hilfe enthält. Möchten Sie das Thema stattdessen in einem normalen Hilfefenster sehen? Dann müssen Sie eine Callback-Prozedur verwenden, weil ansonsten -wie oben!- die Nachricht "`WM_HELP`" an das Besitzerfenster geschickt wird. Und weil das Programm in dem Fall eine Kontext-ID übermittelt bekommt, reagiert es natürlich entsprechend.
Die Callback-Prozedur sieht wie folgt aus:

```
procedure MsgBoxCallback(phi: PHelpInfo); stdcall;
begin
  URL := szHelpFile + '>HLPWIN';
  WinHelp(phi.hItemHandle,@URL[1],HELP_CONTEXT,phi.dwContextId);
end;
```

Wir legen hier lediglich das Zielfenster fest und benutzen den üblichen Aufruf von "`WinHelp`" zur Anzeige eines einzelnen Themas. Zur Verwendung dieser Umleitung setzen wir die `lpfnMsgBoxCallback`-Membervariable des `TMsgBoxParams`-Records vor dem Aufruf von "`MessageBoxIndirect`" auf unsere Prozedur:

```
msgbox.lpfnMsgBoxCallback := @MsgBoxCallback;
```

Damit erscheint das gewünschte Hilfethema wie gewohnt in einem eigenen Fenster. Das Beispielprogramm demonstriert das mit Hilfe des Compilerschalters "`NOPOPUP`", der normalerweise aktiv ist und nach obigem Muster das Hilfethema in einem eigenen Fenster anzeigt. Wenn Sie allerdings die Anweisung

```
{$DEFINE NOPOPUP}
```

mit einem Punkt sozusagen "deaktivieren", wird die Hilfe-Information als Popup gezeigt.

7.1.10. Die kontextsensitive Hilfe verwenden

Um in der API-Programmierung die kontextsensitive Hilfe nutzen zu können müssen wir den jeweiligen Elementen zuerst eine Kontext-ID zuordnen. Dazu verwenden wir den Befehl "`SetWindowContextHelpId`". Da wir ja wissen, dass auch Buttons, Eingabefelder usw. im Prinzip nur Fenster sind, dürfte es uns nicht schwer fallen, unserem ersten Button und unserem Eingabefeld eine solche Kontext-ID zu geben.

```
BOOL SetWindowContextHelpId(
    HWND hwnd,                // Fensterhandle
    DWORD dwContextHelpId     // ID
);
```

Üblicherweise besitzen Fenster mit einer kontextsensitiven Hilfe einen entsprechenden Button in der Titelzeile

Allerdings darf unser Fenster dann keine Minimierungs- und Maximierungsschaltfläche besitzen. Eine "Designschwäche" seitens Microsoft, möchte ich mal sagen. Sei´s drum ... Wenn wir diesen Hilfe-Button benutzen wollen, müssen wir unser Fenster mit einem erweiterten Stil-Attribut erstellen:

```
CreateWindowEx({ erweitertes Attribut --> }WS_EX_CONTEXTHELP, ... );
```

Klicken Sie diesen Button an, ändert sich der Mauszeiger. Damit klicken Sie nun auf das gewünschte Element, zu dem Sie die kontextbezogene Hilfe sehen wollen, und es wird eine "WM_HELP"-Nachricht ausgelöst.
An dieser Stelle kurz zum Nachdenken: Unsere Anwendung reagiert bereits auf diese Nachricht. Wir müssen natürlich auch dafür sorgen, dass sie auf die kontextsensitive Hilfe reagiert. Hier ist allerdings wichtig, in welcher Reihenfolge wir vorgehen. Als Beispiel: die "WM_HELP"-Nachricht wird bisher wie folgt behandelt:

```
if(PHelpInfo(lp)^.hItemHandle = wnd) or
  (GetParent(PHelpInfo(lp)^.hItemHandle) = wnd)
then
  SendMessage(wnd,WM_COMMAND,MAKELONG(IDC_TOCBTN,BN_CLICKED),0)
```

Würden wir jetzt den für die Kontext-Hilfe notwendigen Code einfach anhängen ergäbe sich ein Problem, das mit der Dialogbox aus dem vorigen Kapitel zu tun hat: Diesmal ist es ja nicht das Fenster-Handle, das übergeben wird, sondern es ist das Handle des jeweiligen Controls (ein Button, ein Eingabefeld, usw.). Da sich dieses Control aber **auf** unserem Fenster befindet, wird die o.g. Bedingung wirksam, und die Anwendung zeigt nicht die gewünschte Kontext-Hilfe sondern das Inhaltsverzeichnis oder die erste Seite der Hilfe.
Aus dem Grund sollten Sie immer zuerst prüfen, ob eine Kontext-ID übergeben wurde oder nicht. Das heißt, der Code für die kontextsensitive Hilfe muss zuerst geschrieben werden, womit die erweiterte "WM_HELP"-Behandlung dann so aussieht:

```
WM_HELP:
  if(PHelpInfo(lp)^.dwContextId > 0) then
    WinHelp(wnd,@szHelpFile[1],HELP_CONTEXTPOPUP,
    PHelpInfo(lp)^.dwContextId)
  else
  if(PHelpInfo(lp)^.hItemHandle = wnd) or
    (GetParent(PHelpInfo(lp)^.hItemHandle) = wnd)
  then
    SendMessage(wnd,WM_COMMAND,MAKELONG(IDC_TOCBTN,BN_CLICKED),0)
  else
    Result := DefWindowProc(wnd,uMsg,wp,lp);
```

```
typedef struct tagHELPINFO {
    UINT      cbSize;         // Größe des Records
    int       iContextType    // Typ (Fenster oder Menüitem)
    int       iCtrlId;        // Id des Elements
    HANDLE    hItemHandle;    // Handle des Elements
    DWORD     dwContextId;    // Kontext-ID
    POINT     MousePos;       // Mausposition
} HELPINFO
```

7.1.10.1. Direkthilfe via Popupmenü

Widmen wir uns noch diesem Problem. Sie kennen das vielleicht aus einigen Anwendungen, dass Sie - neben dem Hilfe-Button in der Titelleiste - ein Element auch mit der rechten Maustaste anklicken können. Ein Menü erscheint, üblicherweise mit dem Eintrag "Direkthilfe", das Ihnen ebenfalls die Kontext-Informationen anzeigt.
Allzu schwer ist es zum Glück nicht. Für unser Menü fangen wir zunächst die Nachricht "WM_CONTEXTMENU" ab. Im WPARAM-Wert finden wir dann das Fenster-Handle, bei dem diese Nachricht ausgelöst wurde. Auf die Weise können wir bestimmen, wo das Popupmenü erscheinen soll:

```
WM_CONTEXTMENU:
  if(wp = hTocBtn) or (wp = hIdxBtn) then
    begin
      // Popupmenü erzeugen

      // Kontext-Id holen
      cId := GetWindowContextHelpId(wp);
    end
  else
    cId := -1;
```

In dem Fall sind es nur zwei Buttons, bei denen wir das Menü anzeigen lassen. Sie sollten Eingabefelder u.ä. bitte generell ausklammern, da diese ein eigenes Kontextmenü zur Textbearbeitung (Kopieren, Ausschneiden, Einfügen usw.) besitzen. Für solche Elemente stehen Ihnen nach wie vor der Hilfe-Button in der Titelleiste und die F1-Taste zur Verfügung.

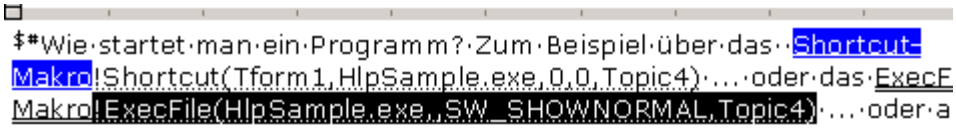
Die Anzeige der Kontext-Information erfolgt dann über die normale Abfrage des Menüeintrags:

```
WM_COMMAND:
  case HIWORD(wp) of
    BN_CLICKED:
      case LOWORD(wp) of
        IDC_DIRECTHELP:
          if(cId > 0) then
            WinHelp(wnd,@szHelpFile[1],HELP_CONTEXTPOPUP,cId);
      end;
  end;
```

7.1.11. Programme starten

Möchte man aus der Hilfe heraus ein Programm starten, bieten sich dafür entsprechende Makros an. Diese Funktion kann übrigens hilfreich sein, wenn man Aufforderungen wie "Öffnen Sie die Optionen des Programms" in der Hilfe mit dem entsprechenden Programm verknüpft usw.

Die Makros werden innerhalb des Textes der RTF-Datei angegeben. Das Prinzip entspricht den Sprungmarken: man definiert einen Text, dessen Unterstreichung später den Link signalisiert, während das Makro selbst unmittelbar als unsichtbarer Text folgt. Beispielsweise:



Zu beachten ist das Ausrufungszeichen. Ohne dieses würde die Makro-Anweisung als Sprungziel gedeutet werden, was zu einer Fehlermeldung des Compilers führt. Als Anweisungen stehen zur Verfügung:

```
ExecFile(program[,        // Programm, das gestartet werden soll
         arguments[,      // evtl. Kommandozeilenparameter
         display-state[,  // SW_SHOW, SW_HIDE o.ä.
         topic-ID]]])     // kann das Programm nicht gestartet
                          // werden, wird das Hilfethema
                          // gezeigt, das der ID (= #-Fußnote) zugeordnet ist


ShortCut(window-class,    // Klassenname des Programms
         program[,        // Dateiname
         wParam[,
         lParam[,
         topic-ID]]])     // Hilfethema im Fehlerfall
```

7.1.11.1. Dokumente drucken und ähnliche Sonderfälle

Als drittes Makro bietet sich auch "`ShellExecute`" an. Allerdings schreibt z.B. Dieter Bremes, in "Delphi 2 - Das Buch" (Kapitel "Hilfeprogrammierung"), dass dieses Makro z.B. nicht funktioniert, wenn man alle Parameter verwendet (und das wären insgesamt 6). Es soll nur funktionieren, wenn man lediglich die ersten drei Parameter verwendet. Aber selbst in dem Fall habe ich das Makro nicht zur Mitarbeit bewegen können.
Ich erwähne es also nur der Vollständigkeit halber und demonstriere stattdessen den Vorschlag von Dieter Bremes, wie man gleich den API-Befehl "`ShellExecute`" verwenden kann:

Da es ein gleichnamiges Makro gibt, kann man den API-Befehl nicht mehr direkt einbinden. Wir können stattdessen den Ansi-Namen der Funktion benutzen. Dazu klicken wir im Helpworkshop auf den Button "CONFIG" und wählen "Add ..." und schreiben:

```
RegisterRoutine("shell32.dll","ShellExecuteA","USSSSi")
```

Die so importierte Funktion lässt sich nun wie ein normales Makro verwenden, wobei natürlich die Syntax des API-Befehls zu beachten ist:

Der importierte Befehl kann nun auch zum Starten von Dokumenten und Programmen verwendet werden. Ebenso ist es aber auch möglich, gezielte Aktionen (wie "print" usw.) auszulösen, die dem System allerdings bekannt sein müssen und vom benutzten Dateityp abhängig sind.

7.1.11.2. Systemsteuerungsmodule ausführen

Für die CPL-Dateien der Systemsteuerung gibt es ein eigenes Makro:

```
ControlPanel(CPL_name[,       // Name des CPL-Moduls
             panel_name,      // Bezeichnung
             tabnum])         // Registerkarte
```

Folgendes ist anzumerken: laut Hilfe des Helpworkshop muss die Bezeichnung - sofern man sie nutzt - der Symbolunterschrift des Moduls entsprechen, also beispielsweise:

Um das Modul "Anzeige" zu starten, müsste demnach das Makro wie folgt aussehen:

```
!ControlPanel(desk.cpl,Anzeige,2)
```

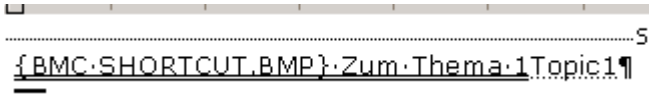
Und das dürfte spätestens auf anderssprachigen Versionen von Windows nicht mehr funktionieren. Aber interessanterweise klappt der Aufruf des CPL-Moduls auch ohne die Bezeichnung problemlos:

```
!ControlPanel(desk.cpl,,2)
```

7.1.12. Helpworkshop-Bitmaps im Text

Der Helpworkshop enthält ein paar Grafiken, die man innerhalb des Textes verwenden kann. So lassen sich z.B. Sprungziele auch durch Grafiken angeben. Die vorhandenen Bitmaps findet man in der Hilfe zum Helpworkshop unter dem Stichwort "bitmaps, supplied by Help Workshop".

Zur Anzeige wird das "`BMC`"-Makro verwendet, das durch die Unterstreichung als Text für ein Sprungziel benutzt werden kann, beispielsweise:



7.1.13. Browse-Sequenzen definieren

Unter einer Browse-Sequenz versteht man im Allgemeinen, dass man mit zwei zusätzlichen Buttons im Hilfe-Betrachter vom aktuellen zum nächsten Thema wechseln kann, und wieder zurück ...

Die Umsetzung einer solchen Sequenz ist relativ einfach. Alle Hilfethemen, die Teil der Sequenz sind, erhalten zusätzlich die Fußnote **+**, der ein Bezeichner zugeordnet wird. Wenn dieser Bezeichner numerisch ist, hat man so Einfluss auf die Präsentation der einzelnen Themen. Die beiliegende Beispiel-Hilfedatei besitzt z.B. folgende Sequenz:

```
001         Seite #1
003         Seite #2
002         Seite #3
```

Was passiert? - Würde man die Browse-Sequenz ausprobieren, würde zunächst die erste Hilfeseite, dann die dritte und zum Schluss erst die zweite angezeigt werden.

Damit das auch funktioniert, aktiviert man in den Fenstereigenschaften des Hilfeprojektes unter "Buttons" die Option "Browse", damit die beiden zusätzlichen Buttons später auch zu sehen sind. Wer kein Fenster definiert hat, fügt das Makro "`BrowseButtons()`" in den Projekteinstellungen über den Button "CONFIG" ein.

Mehr ist nicht erforderlich. Es sei denn, man wünscht unterschiedliche Browse-Sequenzen. In diesem Fall sollte man der numerischen Angabe einen Namen, eine ID oder ähnliches voranstellen und durch Doppelpunkt von der Ziffernfolge abgrenzen.

Alle Sequenzen mit der selben ID gehören demnach zusammen, beispielsweise:

```
main:001
main:002
```

und

```
sub:001
sub:002
```

würden zwei vollkommen verschiedene Sequenzen definieren.

7.2. CHM-Hilfedateien

7.2.1. Vorwort

Bei CHM-Hilfedateien handelt es sich im Prinzip um einfache HTML-Dateien, die mit einem Inhaltsverzeichnis und einem Index zu einer Datei kompiliert werden. Als Entwicklungsumgebung kann man demnach seinen bevorzugten HTML-Editor verwenden und wird so wahrscheinlich eher Erfolge erzielen als bei den alten Hilfedateien. HTML-Kenntnisse sind nicht zwingend erforderlich, helfen aber.

Um CHM-Dateien erstellen zu können, benötigen wir den HTML Helpworkshop, den es kostenlos von Microsoft gibt. Neben dem HHW gibt es Freeware und kommerzielle Tools, mit denen man ebenfalls Hilfedateien erstellen kann, aber für dieses Tutorial beziehen wir uns auf das Programm von Microsoft.

Da wir als Delphi-Programmierer natürlich daran interessiert sind, die neuen Hilfedateien in unseren Programmen zu nutzen, brauchen wir Zugriff auf die speziellen API-Funktionen der CHM-Hilfe. Als die für mich persönlich beste Umsetzung möchte ich das HtmlHelp-API von den JEDIs nennen, das beim Testen so gut wie keine Probleme verursacht hat. Das soll keine (Ab-)Wertung gegenüber anderen Möglichkeiten darstellen. Ich bevorzuge eben das JEDI-API. Wer z.B. mit dem "HTML Help Kit for Delphi" arbeiten möchte, den verweise ich hiermit auf das Tutorial von Martin Strohal.

Doch zurück zum Thema: JEDI. Auf deren Homepage gibt es ein spezielles ZIP-Archiv zum Download, dass nur das HtmlHelp-API und einige Beispiele enthält.
Wir machen trotzdem ein eigenes. In Ordnung?

Hinweis
Diesem Tutorial liegt eine speziell angepasste Version der Unit "`HtmlHlp.pas`" bei, die auf nonVCL-Anwendungen zugeschnitten ist. Es gibt zur offiziellen Version der JEDIs keinen Unterschied, lediglich Registry- und Dateizugriffe wurden mit API-Aufrufen realisiert, um auf die Units "`SysUtils.pas`" und "`Registry.pas`" verzichten zu können.

Einige benutzte Codeteile stammen aus den Beispielen der JEDIs. Ich sagte ja, dass ich diese API-Umsetzung für eine der besten halte. Und damit lässt sich demzufolge auch am besten arbeiten.

7.2.2. Ein kurzes Wort zur HTML-Syntax

Weil dies kein HTML-Tutorial ist, möchte ich mich nur ganz kurz fassen: Wenn Sie noch keine Erfahrung mit dem HTML-Format haben, dann sollten Sie zum Testen Programme wie FrontPage benutzen, in denen Sie am Anfang ohne Code auskommen und Ihre HTML-Seite wie in einer Textverarbeitung gestalten können. So können Sie sich mit den Grundlagen vertraut machen und Ihre Dateien später auch optimieren.

Technisch können Sie all das benutzen, was Sie aus dem Web her kennen (CSS, Grafiken, Skripte, Animationen ...). Wenn Sie z.B. eine Homepage haben und die Dateien lokal auf Ihrem Rechner gespeichert sind, könnten Sie zu Testzwecken sogar diese Homepage in eine CHM-Hilfedatei "verpacken". Für unser Beispiel gestalten wir ein paar einfache HTML-Seiten als Grundlage, wobei wir uns am Inhalt unserer HLP-Beispieldatei orientieren wollen.

Beispiel

```
<html>
<head>
<title>CHM-Hilfedatei, Seite 2</title>
<link rel="stylesheet" type="text/css" href="docstyle.css">
</head>
<body>
<p align="justify">Hier ist die zweite Seite meiner Hilfedatei, die nur einen
  <a href="page1.htm">Verweis</a> auf die erste Seite enth&auml;lt.<br></p>
</body>
</html>
```

Hinweis
Wenn Sie Grafiken z.B. durch Skriptbefehle auswechseln lassen oder z.B. über CSS-Dateien laden, dann kann es passieren, dass die gewünschte Grafik später fehlt. Das liegt daran, dass der CHM-Compiler nur Dateien

berücksichtigt, die in den HTML-Dateien angegeben wurden. Er "weiß" nicht, dass einige Grafiken zur Laufzeit ausgetauscht werden.
Sie können das Problem beispielsweise mit einer leeren HTML-Seite lösen, die nur die Grafiken enthält, ansonsten aber nicht verwendet wird. Der Compiler prüft nicht, ob diese leere Seite auch wirklich benutzt wird; er kompiliert sie aber mit, und damit sind die benötigten Grafiken in Ihrer CHM-Datei enthalten.

Eine zweite Möglichkeit ist es, die gewünschten Dateien in die Dateiliste des Projektes aufzunehmen. Dazu müssen Sie den Auswahlfilter "`*.html`" manuell auf die entsprechende Dateiendung umstellen. Sind die Dateien Teil des Hilfe-Projektes, werden sie auch vom Compiler berücksichtigt.

7.2.2.1. Links zu anderen CHM-Dateien

Wenn Sie auf ein Thema linken wollen, das sich in einer anderen CHM-Hilfedatei befindet, müssen Sie der Datei, auf die Sie zugreifen wollen, einen speziellen Befehl voranstellen: "`ms-its:`" oder "`mk:@MSITStore:`". Danach geben Sie dann die Hilfedatei und das gewünschte Thema an, beispielsweise:

```
<a href="ms-its:System.chm::/Timer/timer-0001.htm">Tutorial &uuml;ber Timer</a>
<a href="mk:@MSITStore:WindowsUI.chm::/Menu/menu-0001.htm">Men&uuml;-
Tutorial</a>
```

"`ms-its:`" benötigt den IE ab Version 4. Im Zweifelsfall sollten Sie daher "`mk:@MSITStore:`" benutzen.

Wenn bei Ihnen bei der Verwendung von "`ms-its:`" Stylesheets ignoriert werden, dann sollten Sie ein Update der Html-Hilfe vornehmen. Das genannte Verhalten ist ein Bug älterer Versionen.

7.2.3. Ein neues Html-Help-Projekt beginnen

Starten wir nun den HTML Helpworkshop, und beginnen wir mit unserem Projekt. Wenn Sie Zeit und Muße haben, dann starten Sie bitte immer ein neues Projekt. Es wäre nämlich auch möglich, HLP-Hilfedateien zu konvertieren. Der HHW bietet dazu einen Assistenten an. Es funktioniert auch recht gut, nur leider benutzt der HHW eigene Dateinamen und HTML-Headerangaben, so dass es sehr schwer ist, die einzelnen Dateien auseinander zu halten. Außerdem muss man ohnehin einige der RTF-Funktionen aufgeben (teilweise gehen auch Textformatierungen verloren), so dass man sich von vornherein lieber auf die Möglichkeiten von HTML usw. konzentrieren sollte.

Für unser Beispiel beginnen wir also ein völlig neues Projekt ("File/New/Project"):

1. *Convert WinHelp Project* ... Übergehen Sie diesen Schritt.
2. *New Project Destination* ... Wählen Sie einen Zielordner und -namen für die Projektdateien aus. Es kann der selbe Ordner sein, in dem sich auch Ihre HTML-Dateien befinden. Wenn Sie das Beispiel nachvollziehen, entscheiden Sie sich beispielsweise für Ihren "`Eigene Dateien`"-Ordner und nennen Sie die Projektdatei z.B. "`test`".
3. *Existing Files* ... Wenn Sie nach dem Muster des letzten Kapitels bereits HTML-Seiten erstellt haben, dann aktivieren Sie hier die Option "HTML files". Der Assistent zeigt Ihnen danach einen Dialog an, in dem Sie alle erstellen HTML-Dateien angeben können. Diese werden Ihrem Projekt automatisch hinzugefügt. Wenn Sie noch keine HTML-Seiten erstellt haben, können Sie diesen Schritt auch später nachholen.
4. *Finish* ... Der Assistent erstellt Ihre Projektdatei.

Bitte holen Sie das Erstellen von Beispiel-HTML-Seiten spätestens jetzt nach und speichern Sie diese am besten im selben Verzeichnis wie Ihre Projektdatei. Ebenfalls möglich ist es, die HTML-Dateien in einem, den Projektdateien untergeordneten Verzeichnis zu speichern. Vorsicht ist geboten, wenn sich die Projektdateien (sprich: HHP-Datei, Index- und Inhaltsverzeichnis) in einem anderen bzw. untergeordneten Verzeichnis befinden, so dass Sie für Links im Inhaltsverzeichnis z.B. einen relativen Pfad wie

```
../some_other_directory/test.htm
```

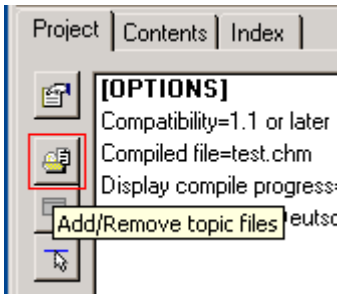
angeben müssten. Auf solche Konstruktionen sollten Sie nach Möglichkeit verzichten. Oder Sie ändern die Angabe im Inhaltsverzeichnis dann manuell mit einem Texteditor z.B. auf

```
test.htm
```

Der Compiler des HTML Helpworkshop meldet dann zwar den Fehler, dass ein Eintrag des Inhaltsverzeichnisses nicht gefunden bzw. aufgelöst werden kann, aber die Datei ist dennoch Teil der Hilfe und wird durch einen Klick auf den entsprechenden Eintrag sogar angezeigt.
Allerdings ist dies kein schöner Stil und muss auch nicht in jedem Fall funktionieren. Daher: verzichten Sie bitte darauf und legen Sie alle notwendigen HTML-Dateien im selben oder in einem, den Projektdateien untergeordneten Verzeichnis an.

Hinweis
Dies betrifft nicht Links zwischen HTML-Dateien! HTML-typische Links zwischen Dateien funktionieren auch in einem Hilfeprojekt mit relativen Pfaden problemlos. Es geht wirklich nur um Angaben im Inhaltsverzeichnis und im Index, die sich auf Dateien in anderen, übergeordneten bzw. in Verzeichnissen gleicher Ordnung beziehen.

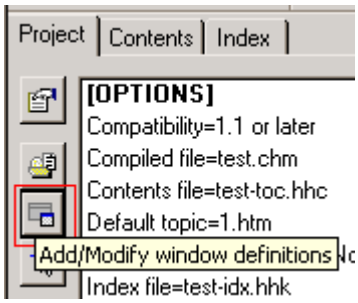
Klicken Sie dann im HTML Helpworkshop auf den im Bild rot markierten Button und wählen Sie Ihre Dateien aus:



Wenn Sie das Projekt nun kompilieren, haben Sie Ihre erste CHM-Datei erstellt. Möglicherweise sieht sie noch nicht besonders schön aus, aber sie funktioniert ...

7.2.4. Das Hilfefenster anpassen

Die Hilfedatei würde sich bisher in einem Standardstil präsentieren, auf den wir allerdings Einfluss nehmen können. Dazu benötigen wir, ähnlich wie bei den HLP-Hilfedateien, ein Fenster. Klicken Sie dazu bitte den entsprechenden Button im HTML Helpworkshop an:



und geben Sie dann einen Bezeichner für das Fenster an. Dieser Name ist nirgends zu sehen, er dient nur internen Zwecken.

Im nun folgenden Dialog können Sie auf mehreren Registerseiten die Grundeinstellungen verändern. Sie können den Stil und die Größe des Fensters beeinflussen. Sie können die Buttons auswählen, die in der Toolbar angezeigt werden sollen. Sie können festlegen, ob das Inhaltsverzeichnis angezeigt werden soll, usw. usw.

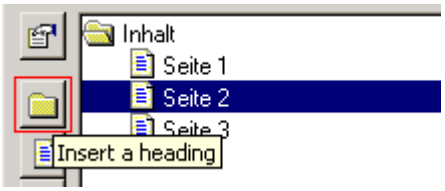
Experimentieren Sie am besten ein wenig mit den Optionen.

7.2.5. Das Inhaltsverzeichnis erstellen

Unsere Beispiel-Hilfedatei zeigt uns bereits eine HTML-Seite an. Dies hat mit den Standardoptionen unseres Projektes zu tun, auf die ich noch eingehen werde. Da wir aber alle unsere HTML-Seiten sehen wollen, benötigen wir ein Inhaltsverzeichnis.

Klicken Sie dazu im HHW auf den Reiter "Contents". Wenn Sie bei der Projekterstellung noch kein vorhandenes Inhaltsverzeichnis ausgewählt hatten, dann erscheint jetzt die Frage, ob Sie ein neues erstellen oder ein vorhandenes öffnen wollen. Wir erstellen also ein neues Inhaltsverzeichnis, dessen Dateiname z.B. "`test-toc.hhc`" lauten könnte.

Um einen neuen Hauptknoten einzufügen (das, was Sie aus den meisten Hilfedateien als Buch-Icon kennen dürften), benutzen wir den Button mit dem gelben Ordner:



Für untergeordnete Einträge, die im Normalfall auf die jeweiligen Themen zeigen, benutzen wir den Button darunter. In beiden Fällen erscheint der selbe Dialog, in dem wir unter "Entry title" den Namen angeben, der im Inhaltsverzeichnis angezeigt werden soll. Mit Hilfe des "Add"-Buttons können wir einem solchen Eintrag dann eine oder auch mehrere HTML-Dateien als Ziel zuordnen.

7.2.5.1. Tipps und Hinweise

Der Eintrag an erster Stelle
Wenn Sie einen Eintrag im Inhaltsverzeichnis haben und einen zweiten hinzufügen wollen, wird die Frage erscheinen, ob Sie den neuen Eintrag am Beginn des Inhaltsverzeichnisses einfügen wollen. Wählen Sie "Nein", wenn der neue Eintrag hinter dem ersten angeordnet werden soll.

Knotenpunkte und Zielverweise
Bei einem Knotenpunkt ist es nicht unbedingt erforderlich, eine HTML-Datei als Ziel anzugeben. Sie können es aber tun, so dass bei einem Klick auf das Thema z.B. die erste, untergeordnete Seite erscheint. Allerdings rate ich von diesem Vorgehen unter bestimmten Bedingungen ab. Ich gehe noch darauf ein. Wenn möglich, dann verwenden Sie bitte eine eigene Seite für die Knotenpunkte, die vielleicht einen thematischen Ausblick auf die untergeordneten Seiten gibt.

Die Images des Inhaltsverzeichnisses
Wenn Sie auch lieber die Buch-Symbole benutzen, dann öffnen Sie die Einstellungen des Inhaltsverzeichnisses (im Bild die Schaltfläche über dem rot markierten Button), und deaktivieren Sie im Dialog die Option "Use folders instead of books". Ohne vorgreifen zu wollen - es gibt eine weitere Möglichkeit, bei der Sie das Inhaltsverzeichnis binär erstellen lassen (bei der späteren Kompilierung der Hilfedatei). In diesem Fall werden ebenfalls alle Symbole ignoriert, und der Hilfebetrachter verwendet die Standardsymbole (in den meisten Fällen also das Buch-Symbol usw.).

Bleiben wir gleich beim Optionen-Dialog des Inhaltsverzeichnisses: auf der Registerseite "Style" haben Sie die Möglichkeit, das Erscheinungsbild des Inhaltsverzeichnisses zu ändern. Standardmäßig sind Optionen aktiv, deren Wirkung Sie durch das Kompilieren Ihrer Hilfedatei erproben können.
Fügen Sie nun z.B. die beiden folgenden Attribute hinzu:

```
Automatically track selection
Only expand a single heading
```

Wenn Sie die Hilfedatei erneut kompilieren, bemerken Sie, dass Ihr Inhaltsverzeichnis jetzt mit einem einfachen Klick geöffnet werden kann. Außerdem kann immer nur ein Knotenpunkt geöffnet sein; die anderen werden automatisch geschlossen. Experimentieren Sie einfach mit den vorhandenen Attributen, wundern Sie sich aber bitte nicht - einige schließen sich gegenseitig aus, d.h. bei der Auswahl einer Einstellung wird evtl. eine andere entfernt, die nicht gleichzeitig genutzt werden kann.

7.2.5.2. Das Inhaltsverzeichnis binär speichern

Microsoft empfiehlt, bei besonders umfangreichen Dateien das Inhaltsverzeichnis binär zu speichern. Beachten Sie bitte, dass dabei die meisten Stildefinitionen verloren gehen. Der Hilfebetrachter nutzt bei einem binären Inhaltsverzeichnis Standardeinstellungen. Sie aktivieren diese Option in den allgemeinen Projekteinstellungen auf der Registerseite "Compiler".

Hinweis

Eine Eigenart (um nicht "Bug" zu sagen) des HTML Helpworkshop ist es, den Eintrag eines Knotenpunktes (= Buches) zu ändern, wenn diesem die HTML-Seite eines untergeordneten Topics zugeordnet ist. Beispiel:

Es handelt sich hier in beiden Fällen um den selben Eintrag im Inhaltsverzeichnis dieses Tutorials. Im oberen Bild besitzt der Knotenpunkt aber einen Verweis auf die erste untergeordnete Seite und erhält merkwürdigerweise dann auch deren Titel. Im unteren Bild hat der Knotenpunkt keinen Verweis und zeigt den korrekten Text an, den Sie auch aus dem Inhaltsverzeichnis kennen.

Dieses Problem tritt allerdings nur auf, wenn Sie das Inhaltsverzeichnis binär speichern und auf eine untergeordnete Hilfeseite verweisen.

Eine Alternative wäre, für den Eintrag eine eigene HTML-Seite zu schreiben, in der alle untergeordneten Themen gesondert aufgeführt werden. In dem Fall wäre der Knotenpunkt nicht von einem dieser untergeordneten Themen abhängig und würde die korrekte Beschriftung behalten.

7.2.6. Einen Index erstellen

Für einen Index gibt es bei CHM-Hilfedateien zwei Möglichkeiten.

7.2.6.1. Der gewohnte Weg über HTML

Im HTML Helpworkshop klicken Sie auf den Reiter "Index" und wählen aus, dass Sie eine neue Datei erstellen möchten. Wenn Sie bereits eine vorhandene Indexdatei haben und nutzen möchten, dann verwenden Sie die entsprechende Option.

Um einen neuen Eintrag einzufügen, benutzen Sie die Schaltfläche mit dem Symbol des Schlüssels. Die Vorgehensweise entspricht dann in etwa dem Anlegen des Inhaltsverzeichnisses: Unter "Keyword" geben Sie den Begriff ein, der im Index erscheinen soll, und mit dem "Add"-Button fügen Sie eine oder mehrere HTML-Seiten als Verweise ein.

Natürlich sind auch gezielte Angaben mit Ankern ("Bookmarks") möglich. Wenn sich das gesuchte Thema beispielsweise etwas weiter unten auf der HTML-Seite befindet und möglicherweise nicht von Anfang an im Hilfefenster zu sehen ist, dann definieren Sie einen Anker an der Position, an der das Thema beginnt.

```
<a name="MeinAnker"></a>
<h2>Mein gesuchtes Thema</h2><br>
<p>bla bla bla</p>
```

Im Index müssen Sie dann den Namen der HTML-Seite und den Namen des Ankers als Verweis angeben, z.B.:

```
Keyword : Ziel mit Anker
Url     : MeineSeite.html#MeinAnker
```

Mehrere Referenzen auf einen Eintrag im Index
Wenn Sie einem Begriff mehrere HTML-Seiten zuordnen, erscheint später in der CHM-Hilfedatei eine Dialogbox mit allen Links zur Auswahl.

Auswahl untergeordneter Einträge erzwingen
Wenn Ihr Begriff ein übergeordneter Index ist, der mehrere Themen gruppiert, dann können Sie die Auswahl eines untergeordneten Eintrags erzwingen. Dazu aktivieren Sie in den Eigenschaften des Eintrags die Option "Target is another keyword" und geben als URL den selben Text an, der auch Ihren Index bezeichnet. Als Titel schreiben Sie "untitled". Bei der Auswahl würde später die Aufforderung erscheinen, einen untergeordneten Eintrag auszuwählen, und genau das wollten wir ja.

7.2.6.2. KLinks

Ohne externe Indexdatei können Sie Indexeinträge auch mit Hilfe von KLinks anlegen. Dabei handelt es sich um ein ActiveX-Objekt, das in den HTML-Code der Seite eingefügt wird. Sie öffnen die gewünschte HTML-Seite im HTML Helpworkshop und positionieren den Cursor an der Stelle im Code, an der Sie die KLinks einfügen wollen. Dann benutzen Sie den Menübefehl "Edit/Compiler Information ...". (Dieser Befehl ist nur verfügbar, wenn Sie eine HTML-Seite geladen haben.)
Benutzen Sie nun auf der Registerseite "Keywords" den Button "Add" und tragen Sie die gewünschten Worte ein, die später im Index erscheinen sollen. Mehrere Worte werden durch Semikolon voneinander getrennt.

Wenn Sie alle Worte eingetragen haben, dann fügt der HHW der Seite z.B. folgenden Code hinzu:

```
<object type="application/x-oleobject"
  classid="clsid:1e2a7bd0-dab9-11d0-b93a-00c04fc99f9e">
    <param name="Keyword" value="anzeigen">
    <param name="Keyword" value="Delphi">
    <param name="Keyword" value="erster Hilfetext">
</object>
```

Mehrere Referenzen auf einen Eintrag im Index
Hier gilt das selbe wie bei dem Weg über die externe Indexdatei: wenn Sie einen Begriff mehrfach in verschiedenen HTML-Seiten benutzen, erscheint bei der Auswahl eine Dialogbox mit allen Entsprechungen.

7.2.6.3. KLinks innerhalb der Hilfedatei verwenden

Mit KLinks werden normalerweise die gefundenen Themen aufgelistet. Dazu öffnen Sie die Seite, die einen solchen Suchverweis enthalten soll, wieder im HTML Helpworkshop und positionieren den Cursor an einer gewünschten Stelle innerhalb des "`BODY`"-Tags. Klicken Sie nun auf den "HTML Help ActiveX Control"-Button in der Toolbar und wählen Sie "Keyword Search" aus der oberen Liste aus. Die untere Eingabezeile enthält einen eindeutigen Namen für das Control. Wenn Sie mehrere verwenden wollen oder müssen, muss jedes Control einen eigenen und eindeutigen Namen haben.

Im nächsten Schritt können Sie auswählen, ob das Control als Button erscheinen oder unsichtbar sein soll. Wenn Sie einen Button als Control gewählt haben, können Sie im nächsten Schritt über dessen Gestaltung befinden.

Dann geben Sie die Worte an, nach denen gesucht werden soll. Mehrere sind wieder durch Semikolon voneinander zu trennen. Befinden sich die Worte in einer anderen Hilfedatei, können Sie diese hier ebenfalls angeben. Der HHW fügt letztlich z.B. folgenden Code ein:

```
<OBJECT id="kwrd" type="application/x-oleobject"
  classid="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11"
  codebase="hhctrl.ocx#Version=4,74,8875,0" width=100 height=100>
    <PARAM name="Command" value="KLink">
    <PARAM name="Button" value="Text:Suche nach dem doppelt genutzten Eintrag">
    <PARAM name="Flags" value=",,1">
    <PARAM name="Item1" value="">
    <PARAM name="Item2" value="doppelt genutzter Eintrag">
</OBJECT>
```

Wenn Sie sich für ein unsichtbares Control entschieden haben, dann können Sie die Funktion mit Hilfe eines Scriptlinks auslösen

```
<a href="javascript:kwrd.Click()">Suche nach dem doppelt genutzten Eintrag</a>
```

Als schöner Nebeneffekt lässt sich hier auch eine Seite festlegen, die im Fehlerfall angezeigt wird. Im o.g. Code des ActiveX-Controls ist das nicht erforderlich, weil der Button automatisch deaktiviert wird, wenn das Thema nicht gefunden wurde. Dafür sorgt die Zeile "`<PARAM name="Flags" ...>`". Bei einem unsichtbaren Control, bzw. einem Button-Control ohne das eben genannte Flag, ergänzen Sie den Parameter "`DefaultTopic`", dem Sie als Wert den Namen der gewünschten Seite übergeben.

```
    <PARAM name="DefaultTopic" value="oops.htm">
```

Die Beispielhilfedatei, die diesem Tutorial beiliegt, macht es anhand eines unbekannten Begriffs vor.

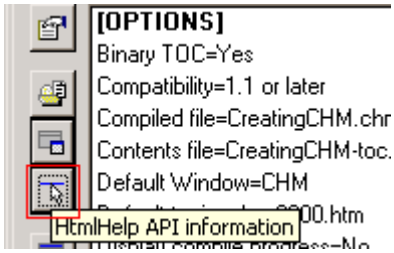
7.2.7. Texte für die kontextsensitive Hilfe

Die kontextbezogene Hilfe ist Ihnen aus vielen Anwendungen und auch von unserer HLP-Beispieldatei bekannt. Mit den HLP-Dateien war es leichter zu lösen (auch dank der Unterstützung dieses Hilfetyps durch Delphi), aber die Textpopups der neuen CHM-Hilfe sind auch nicht allzu kompliziert.

Sie haben zwei Möglichkeiten: Sie können die Popupinformationen in Ihrem Programm unterbringen und eine Fähigkeit der HTML-Hilfe zur Anzeige nutzen, oder Sie erstellen eine Textdatei mit den Informationen, die Sie in Ihre CHM-Hilfedatei integrieren und in Ihrem Programm aufrufen. Da dies der schwere Weg ist, gehen wir ihn. Unsere Datei ist eine ganz normale Textdatei, die folgenden Aufbau haben sollte:

```
.TOPIC 1000
Dies ist mal nur ein Popup-Text.

.TOPIC 1002
Ein Editfeld, man höre und staune ...
```

Um diese Datei in unser Hilfeprojekt aufzunehmen, fügen wir sie einmal wie eine HTML-Datei hinzu. Der Filter der Dateiauswahl muss dazu manuell auf "`*.*`" gestellt werden. Im zweiten Schritt klicken wir auf den Button "HtmlHelp API Information":



und wählen unter "Text-Popups/Text file" die eben erstellte Datei mit den Popups ein zweites Mal aus. Wenn Sie die CHM-Hilfedatei jetzt neu kompilieren, ist die Textdatei enthalten.

7.2.8. Ein HLP-Projekt konvertieren

In diesem Kapitel zeige ich, wie man mit Hilfe der Projekt- und Quelldateien einer HLP-Hilfedatei automatisch die notwendigen Dateien für eine CHM-Hilfedatei generieren lassen kann. Als Beispiel habe ich mich für die "`Sample.hlp`" entschieden, weil Sie deren Projektdateien als ZIP-Archiv im Ordner des HLP-Beispielprogramms finden.

Entpacken Sie dieses Archiv bitte in einen Ordner Ihrer Wahl und kontrollieren Sie, dass drei Dateien vorhanden sind: die Projektdatei (*.hpj), das Inhaltsverzeichnis (*.cnt) und der eigentliche Hilfetext (*.rtf). Stellen Sie bei Ihren eigenen Projekten bitte sicher, dass sich alle notwendigen Dateien an den vorgesehenen Orten befinden.

7.2.8.1. Warum und wann sollte man konvertieren?

Nützlich ist diese Funktion, wenn Sie noch wenig Erfahrung mit CHM-Hilfedateien haben, sich dafür aber besser mit den HLP-Dateien auskennen. Ebenfalls hilfreich kann sie sein, wenn Sie wenig Erfahrung mit HTML haben. Will man sich an die Arbeit mit dem HTML Helpworkshop gewöhnen, ist die Konvertierungsfunktion durchaus empfehlenswert.

7.2.8.2. Der Konvertierungsassistent

1. Starten Sie bitte den HTML Helpworkshop und wählen Sie den Menübefehl "New/Project". Aktivieren Sie bitte im ersten Dialogfenster die Option "Convert WinHelp project".
2. Wählen Sie nun bitte im oberen Eingabefeld die Projektdatei (*.hpj) der alten HLP-Datei aus, und geben Sie im unteren Teil Verzeichnis und Namen des neuen Hilfeprojektes an, das erzeugt werden soll.
3. Klicken Sie auf "Fertigstellen" und warten Sie. Abhängig vom Umfang Ihrer HLP-Datei, bzw. der entsprechenden Projektdateien, dauert die Konvertierung einen Augenblick.

7.2.8.3. Nachteile der Konvertierung

Willkürliche Wahl von Dateinamen
Das ist noch verschmerzbar. Schöner wäre sicher gewesen, wenn der HHW die Seiten nach dem Auftreten im RTF-Text nummeriert, etwa "`Sample-0001.htm`" usw., aber ein kurzer Blick in die Datei verrät einem ja, welches Thema enthalten ist.

"(Untitled)"
Das Problem ist schon gravierender. Obwohl in der Beispiel-HLP-Datei die Fußnote für Überschriften verwendet wurde, nutzt der HHW bei der Konvertierung für alle HTML-Seiten den gleichen, nichts sagenden Titel. Es wäre auch annehmbar gewesen, den Dateinamen als Titel zu verwenden. So aber wird die Identifizierung der Dateien im HHW unnötig erschwert, weil man in der Auswahlliste erst einmal nur die HTML-Titelangabe und nicht sofort den Dateinamen sieht. Im Fall unseres Beispiels ist es noch tolerierbar, weil es sich lediglich um 6 konvertierte HTML-Seiten handelt, bei größeren Projekten dürfte sich das aber zum Hauptärgernis entwickeln.

Popup-Sprungmarken
Interne Popups werden nicht als solche erkannt sondern als separate HTML-Seiten (u.U. ohne Eintrag im Inhaltsverzeichnis) angelegt. Ausgehend davon sind die Informationen dieser HTML-Seiten zu verwenden und wie beschrieben in CHM-Popups umzuwandeln.

Kontexthilfe
Die Texte der kontextsensitiven Hilfe werden ebenfalls als separate HTML-Seiten angelegt, weil sie im RTF-Text ja auch separate Seiten waren. Durch die Änderung des Hilfeformats sind die Informationen der Kontexthilfe nach dieser Anleitung anzupassen.

Programme starten
Im Fall meiner Beispieldatei wurde nur das "`Shortcut`"-Makro als solches erkannt und in die CHM-Entsprechung konvertiert. Sie müssen ggf. die originale HLP-Datei aufrufen und kontrollieren, ob und welche Makros Sie verwenden und ob diese korrekt konvertiert worden sind.

7.2.9. Die Hilfe verwenden

Wir brauchen nun die schon angesprochenen Units der JEDIs. Davon binden wir die Unit "HtmlHlp.pas" in unser Projekt ein.

Probleme vermeiden
Ganz wichtig ist es, den Compilerschalter HTMLHELP_DYNAMIC_LINK zu benutzen. Damit wird versucht, die CHM-Funktionen dynamisch zu laden. Wenn Sie den Schalter nicht nutzen, wird davon ausgegangen, dass CHM-Hilfedateien generell unterstützt werden. Das kann auf Windows 95/NT4-Systemen zu Problemen führen, weil diese standardmäßig mit dem CHM-Typ nichts anfangen können. Sie könnten alternativ eine zweite Hilfedatei im HLP-Format schreiben, oder Sie fordern den Anwender auf, sich das HtmlHelp-Update von Microsoft zu installieren, wenn er Ihre Hilfe nutzen möchte.

Durch den o.g. Compilerschalter steht Ihnen die Bool-Variable "HtmlHelpIsAvail" zur Abfrage zur Verfügung. Der Schalter ist Teil einer Include-Datei. Allerdings ist er entweder fehlerhaft eingetragen oder auf diese Weise standardmäßig deaktiviert. Öffnen Sie im Zweifelsfall bitte die Datei "HtmlHlp.inc" und ersetzen Sie den Eintrag HTML_HELP_DYNAMIC_LINK durch den o.g. korrekten Compilerschalter. - Bei der Version, die diesem Tutorial beiliegt, habe ich dies bereits erledigt.

Als Grundlage soll unser HLP-Beispielprogramm dienen, das wir nun für die CHM-Hilfedatei ein wenig ändern. An erster Stelle steht dabei natürlich der Dateiname in der Variablen "szHelpFile", der angepasst werden muss. Wenn das geschehen ist, benutzen wir den neuen Befehl "HtmlHelp" in unserem Programm:

HtmlHelp-Definition

```
HWND HtmlHelp(HWND     hwndCaller,
              LPCSTR   pszFile,
              UINT     uCommand,
              DWORD    dwData);
```

WM_COMMAND:

```
  case HIWORD(wp) of
    BN_CLICKED:
      case LOWORD(wp) of
        IDC_TOCBTN:
          if(IsWindowEnabled(hTocBtn)) then
            HtmlHelp(wnd,@szHelpFile[1],HH_DISPLAY_TOC,0);
        IDC_IDXBTN:
          if(IsWindowEnabled(hIdxBtn)) then
            HtmlHelp(wnd,@szHelpFile[1],HH_DISPLAY_INDEX,0);
        IDC_THEME2BTN:
          if(IsWindowEnabled(hTheme2Btn)) then
            begin
              URL := szHelpFile + '::/page2.htm';
              HtmlHelp(wnd,@URL[1],HH_DISPLAY_TOPIC,0);
            end;
      end;
  end;
```

Der o.g. Codeauszug demonstriert Ihnen anhand des Buttons "IDC_THEME2BTN", wie Sie gezielt ein Thema der Hilfedatei aufrufen können.

Beachten Sie bitte die Syntax der CHM-Hilfe, die sich von der Syntax im Web unterscheidet. Vom Web sind Sie es gewohnt einen Doppelpunkt und zwei Schrägstriche zu verwenden, bei der CHM-Hilfe ist das genau umgekehrt.
Hier liegt übrigens auch eine der Hauptfehlerquellen bei CHM-Hilfedateien. Bei der Kompilierung der HTML-Seiten zu einer CHM-Datei werden standardmäßig alle Ordner- und Dateireferenzen beibehalten. Befindet sich Ihre HTML-Datei also in einem eigenen Verzeichnis, dann müssen Sie dieses Verzeichnis beim Aufruf des Themas mit angeben. Gehen wir als Beispiel davon aus, die Datei "page2.htm" würde nicht im Verzeichnis der Projektdateien sondern stattdessen im untergeordneten Verzeichnis "htdocs" gespeichert sein. Der Aufruf von oben müsste daher so geändert werden:

```
URL := szHelpFile + '::/htdocs/page2.htm';
```

Auch hier sollten wir beim Beenden des Programms ein evtl. offenes Hilfefenster ebenfalls schließen. Das ist nur eine kleine Änderung im "WM_DESTROY"-Teil:

```
WM_DESTROY:
  begin
    if(HtmlHelpIsAvail) then HtmlHelp(wnd,nil,HH_CLOSE_ALL,0);
    PostQuitMessage(0);
  end;
```

Als einzig noch erwähnenswerte Änderung soll noch auf die Sicherheitsprüfung eingegangen werden. Bei unserem HLP-Beispiel genügte es, das Vorhandensein der Hilfedatei zu prüfen. Bei der CHM-Hilfe müssen Sie aber auch damit rechnen, dass ein Anwender Ihr Programm unter Windows 95 oder NT 4 laufen lässt. Ohne das Update von Microsoft gibt es dort keine Html-Hilfe. Daher müssen wir zusätzlich prüfen, ob das dafür zuständige OCX-Control geladen wurde:

```
if(not fileexists(szHelpFile)) or (not HtmlHelpIsAvail) then
  begin
    // Elemente deaktivieren
  end;
```

7.2.9.1. Die Hilfe in einer MessageBox verwenden

Wie bereits bei den HLP-Dateien beschrieben, kann man auch hier "MessageBox(Ex)" benutzen, um die Hilfe aufzurufen. Selbstverständlich lässt sich für die Anzeige von speziellen Themen der Hilfedatei auch wieder "MessageBoxIndirect" verwenden, weil aber die Technik eine andere ist, muss ein wenig nachgeholfen werden.

Um das Thema als Popup anzuzeigen, sollten Sie Ihre Textdatei mit den kontextsensitiven Informationen entsprechend ergänzen:

```
.TOPIC 3000
Wenn Sie diese Seite sehen, dann haben Sie auf den Hilfe-Button
einer MessageBox geklickt. Sie können solche Sachen natürlich
sinnvoller nutzen.
```

(Den gleichen Text können Sie auch auch direkt im Programm unterbringen, wie noch beschrieben werden wird.) Würden Sie die Hilfedatei nun neu kompilieren und ausprobieren, sollte der obige Text eigentlich erscheinen.
Wenn Sie ihn aber auch wieder in einem Hilfefenster sehen wollen, dann sollten Sie zuerst natürlich eine entsprechende HTML-Seite anlegen und Ihrem Hilfe-Projekt hinzufügen. Dann benutzen Sie die Funktion "MsgBoxCallback" aus dem o.g. HLP-Kapitel als Grundlage, ändern jedoch den Aufruf der alten Hilfe für die HTML-Hilfe um:

```
procedure MsgBoxCallback(phi: PHelpInfo); stdcall;
begin
  if(phi.dwContextId = HLP_PAGE_4) then begin
    URL := szHelpFile + '::/page4.htm';
    HtmlHelp(phi.hItemHandle,@URL[1],HH_DISPLAY_TOPIC,0);
  end;
end;
```

In diesem Fall ist eine Abfrage enthalten, damit das gewünschte Thema nur angezeigt wird, wenn die übermittelte Kontext-ID stimmt. Wenn Sie mehrere verschiedene Themen auf diese Weise anzeigen wollen, dann bietet sich ein Array oder vielleicht eine **case**-Abfrage an.
Wie dem auch sei, mehr ist an Änderungen nicht erforderlich, und Ihr Programm ruft nun auch Themen aus CHM-Hilfedateien auf.

7.2.10. Ein Beispiel für VCL-Programmierer

Als Grundlage für dieses Beispiel dient die VCL-Version des HLP-Beispielprogramms. Unser Ziel ist es, die neue CHM-Hilfe einzubinden, ohne zuviel von unserem Projekt zu ändern.

> *Hinweis*
> Grundsätzlich gelten die hier im Text angesprochenen Änderungen und Sicherheitsprüfungen auch für die VCL-Version. Ich werde also nicht noch einmal darauf eingehen und verweise Sie stattdessen auf das entsprechende Kapitel dieses Tutorials.

Es gibt eine sehr schöne Möglichkeit, wie wir unsere neue CHM-Hilfedatei nutzen können. Wir schreiben unser eigenes "**OnHelp**"-Ereignis und fangen dort alle relevanten Nachrichten ab. In unserem Beispiel sind das drei Dinge: wir wollen das Inhaltsverzeichnis anzeigen, den Index und ganz gezielt ein Thema aus der Hilfedatei.

```
type
  TForm1 = class(TForm)
    ...
  private
    { Private-Deklarationen }
    function AppOnHelp(Command: Word; Data: Longint; var CallHelp: Boolean):
Boolean;
  ...


function TForm1.AppOnHelp(Command: Word; Data: Longint; var CallHelp: Boolean):
Boolean;
var
  URL : string;
begin
  CallHelp := false; // WinHelp unterdrücken
  Result   := true;  // Standard-Rückgabewert der Funktion
  URL      := ExtractFilePath(paramstr(0)) + Application.HelpFile;

  case Command of
    HELP_QUIT:
      HtmlHelp(0,nil,HH_CLOSE_ALL,0);
    HELP_FINDER:
      HtmlHelp(0,@URL[1],HH_DISPLAY_TOC,Data);
    HELP_PARTIALKEY:
      HtmlHelp(0,@URL[1],HH_DISPLAY_INDEX,Data);
    else
      Result := false;
  end;
end;
```

Damit sind zwei unserer drei Bedingungen schon erfüllt: Inhaltsverzeichnis ("`HELP_FINDER`") und Index ("`HELP_PARTIALKEY`") unserer CHM-Hilfedatei werden mit diesem Code angezeigt. Im "**OnCreate**"-Ereignis der Form weisen wir dieses selbstgeschriebene Hilfeereignis nun zu, und im "**OnDestroy**"-Ereignis geben wir es wieder frei:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  if(fileexists(ExtractFilePath(paramstr(0)) + Application.HelpFile) = FALSE) or
    (HtmlHelpIsAvail = FALSE) then
  begin
    // Elemente deaktivieren
  end
  else Application.OnHelp := AppOnHelp; // "OnHelp"-Event festlegen
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Application.OnHelp := nil; // "OnHelp"-Event freigeben
end;
```

Im Klartext bedeutet das: wir machen uns nicht die Mühe unseren Code Zeile für Zeile abzusuchen und evtl. alte Hilfebefehle durch die neuen zu ersetzen. Wir fangen sie stattdessen ab, unterdrücken den Aufruf der alten Hilfe und leiten die Kommandos an die neue Html-Hilfe weiter.
Ebenso elegant können wir die einzelnen Hilfethemen aufrufen lassen. Unser Beispiel besitzt zwar nur ein gemapptes Thema, die zweite Seite der CHM-Datei nämlich, aber als Demonstration genügt das vollkommen. Schauen wir uns noch einmal den bekannten Aufruf der Hilfe an:

```
if(Theme2Btn.Enabled) then
  Application.HelpCommand(HELP_CONTEXT,2000);
```

Da wir unser eigenes "**OnHelp**"-Ereignis geschrieben haben, müssen wir dort also lediglich "`HELP_CONTEXT`" berücksichtigen. Der Wert 2000 wird im Parameter "`Data`" angegeben und kann dort abgefragt werden:

```
case Command of
  HELP_CONTEXT:
    if(Data = 2000) then
      begin
        URL := URL + '::/page2.htm';
        HtmlHelp(0,@URL[1],HH_DISPLAY_TOPIC,0);
      end;
end;
```

Wenn Sie mehrere Themen gemappt haben, dann wäre es nach meiner Meinung zweckmäßig, die dazu passenden Hilfeseiten in einem Array unterzubringen. Gehen wir in einem fiktiven Beispiel einfach einmal davon aus, dass wir 4 Themen über Buttons oder Menüpunkte aufrufen können, deren IDs entsprechend von 2000 bis 2003 nummeriert sind. Das Array könnte also wie folgt aussehen:

```
const
  MappedTopics : array[2000..2003]of string =
    ('page2.htm','page3.htm','page4.htm','page5.htm');
```

Der Bereich des Arrays entspricht in dem Fall den IDs, um den Aufwand zu verringern. Die schon gezeigte Abfrage eines Themas könnte nun relativ einfach geändert werden:

```
case Command of
  HELP_CONTEXT:
    begin
      URL := URL + '::/' + MappedTopics[Data];
      HtmlHelp(0,@URL[1],HH_DISPLAY_TOPIC,0);
    end;
end;
```

Würde man nun z.B. das Hilfethema aufrufen wollen, dem die ID 2001 zugeordnet ist, würde an den Namen der Hilfedatei (in der Variablen "`URL`") die dazu passende HTML-Seite aus dem Array angehangen werden

```
C:\LastCrap\Sample.chm::/page3.htm
```

Das wird an das Html-Help-API weitergereicht, die dann das gewünschte Thema anzeigen sollte.

7.2.10.1. Die MessageBox

Bleibt noch die MessageBox-Geschichte, die wir hier natürlich auch umsetzen wollen. Die Grundlage ist recht simpel. Wir verwenden "`MessageDlg`" und geben die ID des gewünschten Themas an:

```
const
  HLP_PAGE_4 = 3000;

procedure TForm1.MsgBoxBtnClick(Sender: TObject);
begin
  MessageDlg('Klicken Sie auf den Hilfe-Button',
    mtInformation,[mbOk,mbCancel,mbHelp],HLP_PAGE_4);
end;
```

Damit das Programm auch darauf reagieren kann, müssen wir in unserem "**OnHelp**"-Ereignis die Kontext-ID (in dem Fall 3000) natürlich auch noch berücksichtigen. Im VCL-Beispiel ist dies etwas eleganter gelöst, hier aber (zur besseren Demonstration) eine ausführliche Anweisung:

```
case Command of
  HELP_CONTEXT:
    if(Data = 3000) then
      begin
        URL := URL + '::/page4.htm';
        HtmlHelp(0,@URL[1],HH_DISPLAY_TOPIC,0);
      end;
end;
```

Das war´s!
Unser Programm verwendet zum Aufrufen der Hilfedatei nun die neuen HtmlHelp-Anweisungen, funktioniert aber sonst wie gewohnt.

7.2.10.2. Hinweis für Delphi 6 und 7

In Delphi 6 und 7 soll der o.g. Weg auf Grund eines Problems bzw. Fehlers nicht funktionieren. Ich selbst arbeite mit Delphi 5 und kann daher zu dem Problem nicht wirklich etwas sagen. Ich kann also nur auf die Infos verweisen, die ich erhalten bzw. gefunden habe.

Wie es aussieht, arbeitet die Hilfe in den genannten Versionen nach einem veränderten Prinzip. Ein Teil der Kommandos wird nicht mehr an das "**OnHelp**"-Ereignis weitergeleitet. Unter http://www.helpware.net/downloads/index.htm kann ein Fix heruntergeladen werden, mit dem sich das Problem beheben lässt.

Mein Dank geht an Lothar, der im Support-Forum auf www.luckie-online.de darauf hinwies.

7.2.11. Die kontextsensitive Hilfe verwenden

Für unsere kontextsensitive Hilfe benötigen wir zunächst eine neue Prozedur:

```
// Copyright www.delphi-jedi.org
procedure ShowHelp(Pt: TPoint; ContextId: Integer);
const
  rect : TRect = (Left:-1;Top:-1;Right:-1;Bottom:-1);
var
  Popup : THHPopup;
  URL   : string;
begin
  URL := szHelpFile + '::/popup.txt';

  if (ContextId > 0) then
    begin
      FillChar(Popup, SizeOf(Popup), 0);
      Popup.cbStruct := SizeOf(Popup);
      Popup.hinst := 0;
      Popup.idString := ContextId;
      Popup.pszText := nil;
      Popup.pt := Pt;
      Popup.clrForeGround := TColorRef(-1);
      Popup.clrBackground := TColorRef(-1);
      Popup.rcMargins := rect;
      Popup.pszFont := '';

      HtmlHelp(0,@URL[1],HH_DISPLAY_TEXT_POPUP,dword(@Popup));
    end;
end;
```

Hier sehen Sie auch, wie die Datei mit den Popup-Texten benutzt werden muss. Durch die Übergabe der Kontext-ID in

```
      Popup.idString := ContextId;
      Popup.pszText := nil;
```

wird das entsprechende Topic aus der Textdatei geladen und entsprechend angezeigt. Wenn sich die Texte für die kontextsensitive Hilfe innerhalb Ihres Programms befinden, müssen Sie ein wenig anders vorgehen. Nehmen wir als Beispiel an, dies wären die notwendigen Informationen:

```
const
  ContextInfo : array[1000..1002]of string =
    ('Dies ist mal nur ein Popup-Text.',
     '', // Dummy
     'Ein Editfeld, man höre und staune ...');
```

Dann benutzen Sie stattdessen diesen Code:

```
      Popup.idString := 0;
      Popup.pszText := pchar(ContextInfo[ContextId]);
```

In dem Fall ist die Angabe der Textdatei auch nicht erforderlich, und Sie brauchen lediglich den Namen der CHM-Hilfedatei angeben. Das Beispielprogramm verdeutlicht das Prinzip mit Hilfe eines Compilerschalters. Wenn Sie das Programm normal starten und nutzen, kommen die Popup-Texte aus der Hilfedatei, wenn Sie stattdessen den Schalter `CONTEXTINSIDE` benutzen, kommen die Texte aus dem o.g. String-Array innerhalb des Programms.

Damit unser Hilfe-Button in der Titelleiste und die F1-Taste ihre Daseinsberechtigung behalten, ändern wir den Code unseres HLP-Beispielprogramms beim Abfangen der Nachricht "`WM_HELP`" und übergeben Position des Mauszeigers und Kontext-ID an die neue Funktion:

```
WM_HELP:
  if(PHelpInfo(lp)^.iContextType = HELPINFO_WINDOW) then begin
    if(PHelpInfo(lp)^.dwContextId > 0) then
      ShowHelp(PHelpInfo(lp)^.MousePos,PHelpInfo(lp)^.dwContextId)
  end else
    Result := DefWindowProc(wnd,uMsg,wp,lp);
```

Lediglich unser Popupmenü muss noch ein wenig ergänzt werden. Da wir uns diesmal selbst um die Anzeige der Kontext-Informationen kümmern, müssen wir auch dafür sorgen, dass sie an der richtigen Stelle erscheinen. Dazu erweitern wir lediglich die Klickfunktion unseres Menüeintrages und holen uns die Position des Mauszeigers:

```
WM_COMMAND:
  case HIWORD(wp) of
    BN_CLICKED:
      case LOWORD(wp) of
        IDC_DIRECTHELP:
          if(cId > 0) then
            begin
              GetCursorPos(p); ShowHelp(p,cId);
            end;
      end;
  end;
```

Den Rest des Programms können wir unverändert lassen.

7.2.11.1. Die VCL-Änderungen

Schauen wir uns an dieser Stelle noch die VCL-Version an. Natürlich benötigen wir die Prozedur "`ShowHelp`" auch hier. Des Weiteren ergänzen wir unsere "`AppOnHelp`"-Funktion wie folgt:

```
function TForm1.AppOnHelp(Command: Word; Data: Longint; var CallHelp: Boolean):
Boolean;
const
  p   : TPoint = (X:0;Y:0);
begin
  case Command of
    ...
    HELP_SETPOPUP_POS:
      p := SmallPointToPoint(TSmallPoint(Data));
    HELP_CONTEXTPOPUP:
      ShowHelp(p,Data);
    ...
  end;
end;
```

Dadurch funktioniert der Hilfe-Button in der Titelleiste schon einmal.

Es bleibt, wie bei der nonVCL-Version, nur eine Erweiterung des Popupmenü-Items, da wir auch hier die Positionierung der Popup-Fenster selbst übernehmen. Im Prinzip nutzen wir eine eben hinzugefügte Funktionalität und senden vor dem Aufruf der Kontext-Hilfe die Nachricht "`HELP_SETPOPUP_POS`" an unser eigenes "**OnHelp**"-Ereignis:

```
procedure TForm1.dHelpClick(Sender: TObject);
var
  p : TPoint;
begin
  if(mThisControl <> nil) and (mThisControl.HelpContext > 0) then
    begin
      GetCursorPos(p);
      Application.HelpCommand(HELP_SETPOPUP_POS,MAKELONG(p.X,p.Y));
      Application.HelpCommand(HELP_CONTEXTPOPUP,mThisControl.HelpContext);
    end;
end;
```

Jetzt funktioniert es auch hier.

7.2.12. Popups innerhalb der Hilfedatei

Auch in CHM-Hilfedateien sind Popups möglich. In etwa ist das mit den Popup-Links der HLP-Hilfedateien vergleichbar. Wir benötigen dazu nur die Objektdeklaration des ActiveX-Objektes:

```
<OBJECT id="popup" type="application/x-oleobject"
  classid="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11">
</OBJECT>
```

und ein wenig JavaScript; eigentlich sind es nur zwei Konstanten:

```
<script language="JavaScript"><!--

        Popup     = "Dies ist mal nur ein Popup-Text"
        TippFont  = "Tahoma,8,,"

//--></script>
```

Aufgerufen wird das dann über einen normalen Link. Den folgenden Auszug finden Sie in der Beispielhilfedatei, die dem Delphi-Programm beiliegt:

```
<p align="justify">Dies wird mein erster Hilfetext, den ich sp&auml;ter in
meinem Delphi-Programm anzeigen
  lassen m&ouml;chte. Hier ist eine Sprungmarke f&uuml;r ein
  <a href="javascript:popup.TextPopup(Popup,TippFont,5,5,-1,-
1)">Popup</a>.<br></p>
```

7.2.13. Programme starten

Auch bei CHM-Hilfedateien können Sie Programme starten lassen. Dazu öffnen Sie die HTML-Seite, die den Startlink enthalten soll, am besten im HTML Helpworkshop und gehen mit dem Cursor an eine beliebige Stelle im Header oder Body:

1. Benutzen Sie nun den Button "HTML Help Active X Control" (das ist der dritte Button mit dem Zauberhut in der unteren Toolbar). Wählen Sie dann aus der Auswahlliste den Punkt "ShortCut" und vergeben Sie in der unteren Eingabezeile einen eindeutigen Namen für das Objekt wenn Sie mehrere Programme über eine HTML-Seite starten wollen.
2. Klicken Sie "Weiter" und wählen Sie aus, ob der Link als Button dargestellt werden oder versteckt sein soll. Letzteres ist sinnvoll, wenn Sie den Link im Text unterbringen wollen, bei dem ein Button vielleicht das Layout zerstören würde.
3. Klicken Sie "Weiter" und geben Sie nun den Namen des Programms, ggf. Parameter und den Namen der Fensterklasse an.
4. Im vierten Schritt können Sie entscheiden, ob und welche Message Sie mit welchen Werten an das Programm schicken möchten. (Diese Möglichkeit ist für Programmierer sicher interessanter als für die, die nur eine Dokumentation schreiben wollen oder müssen.)

Für den Fall, dass das Programm nicht gefunden wurde, können Sie in der unteren Eingabezeile eine HTML-Seite angeben, die dann gezeigt wird.

Wenn Sie auf "Fertigstellen" klicken, wird Ihrer HTML-Seite ein Objekt hinzugefügt, das den Start auslöst.

```
<OBJECT id=NotepadLaunch type="application/x-oleobject"
  classid="clsid:adb880a6-d8ff-11cf-9377-00aa003b7a11"
  codebase="hhctrl.ocx#Version=4,74,8875,0">
    <PARAM name="Command" value="ShortCut">
    <PARAM name="Item1" value="notepad,notepad.exe,">
</OBJECT>
```

Wenn Sie keinen Button sondern einen versteckten Link ausgewählt haben, dann müssen Sie noch für eine passende Klickaktion sorgen, die wie folgt aussehen kann:

```
Klicken Sie <a href="javascript:NotepadLaunch.Click();">hier</a> um Notepad zu
starten.
```

Hinweis
Dieses ActiveX-Objekt lässt sich nur innerhalb der HTML-Hilfe verwenden. Sie müssen also nicht befürchten, dass so ausgerüstete HTML-Seiten im Internet nun Programme auf Ihrem Rechner starten können.