

# LISTVIEW.DLL

Version 2.11 - Freeware  
Letzter Stand: 25.März 2009

Autor:

Frank Abbing  
Margarethenstr.13a  
48683 Ahaus

Email: [frabbing@gmx.de](mailto:frabbing@gmx.de)

Webpage: <http://frabbing.bplaced.net>

## Einführung:

Was ist denn überhaupt ein Listview? Ob sie's glauben oder nicht, sobald sie ihren Windows-PC anschalten, erscheint schon das erste Listview... wo, fragen sie?

Auf ihrem Bildschirm... Der Windows Bildschirm ist nämlich nichts anderes als ein Listview!!!

Allerdings kein Report-Listview. Aber genau diese Report Listview's unterstützt die Listview.dll. Reports, das sind Tabellen.

Mit der Listview.dll können also mehrspaltige Tabellen, einfach, komfortabel und effektiv genutzt werden.

So können sie z.B. mit Leichtigkeit eine automatische Sortierung aller Einträge der Tabelle erreichen, basierend auf den Werten jeder einzelnen Spalte. Auf Knopfdruck kann das Listview auf- oder absteigend sortiert werden, nach Zahlen oder nach Buchstaben.

Die Sortierung nach Zahlen funktioniert auch mit negativen und dezimalen Zahlen (z.B. -2517 oder -1736,59376 usw.). Es können Icons (kleine Bilder) eingebaut werden, oder auch nicht, Trennstriche sind wählbar, freie Farbwahl von Text und Hintergrund.

Drag & Drop sorgt für ein komfortables Verschieben von Texten, oder ein im- und exportieren derselben.
































Aber auch ganze Dateien (.csv oder .dbf) können in einem Rutsch schnell in das Listview eingelesen werden, und auch wieder exportiert werden.

Komplette Spalten und auch Zeilen können extra eingefärbt werden, und der Hintergrund kann mit einer Grafik verschönert werden. Checkboxes lassen den Anwender bestimmte Zeilen markieren. Editfelder sorgen für ein anwenderfreundliches Editieren aller oder ausgesuchter Einträge (mehr dazu weiter unten).

Natürlich kann das Listview auch komfortabel zu Papier gebracht werden..

Vergessen sie ruhig Listboxen, auch die sortierten. Listviews sind deutlich flexibler, schneller und ansprechender, dank der Listview.dll mit ihren fast 100 Power-Funktionen!

Nur, wie sieht überhaupt ein Listview aus? Hier ein Beispiel:

left-aligned	left-aligned	left-aligned	center	right-aligned
<input checked="" type="checkbox"/>  test1	 test1	 test1	test1	 test1
<input checked="" type="checkbox"/> test2	test2	 test2	 test2	test2
<input type="checkbox"/>  test3	test3	test3	<input type="checkbox"/>	test3
<input type="checkbox"/>  test4	 test4	test4	 test4	 test4
<input checked="" type="checkbox"/> test5	 test5	 test5	 test5	<input type="checkbox"/> test5
<input type="checkbox"/> test6	 test6	 test6	<input type="checkbox"/> test6	 test6
<input type="checkbox"/>  test7	<input type="checkbox"/> test7	<input type="checkbox"/> test7	 test7	 test7
<input checked="" type="checkbox"/>  test8	<input type="checkbox"/> test8	 test8	<input type="checkbox"/> test8	test8
<input type="checkbox"/> test9	 test9	test9	 test9	 test9
<input checked="" type="checkbox"/> test10	 test10	test10	 test10	 test10
<input type="checkbox"/> test11	test11	 test11	 test11	 test11

Ein Listview ist also eine Tabelle mit variabler Spalten- und Zeilenanzahl.

Hier eine kurze Erklärung einiger Ausdrücke, die in dieser Anleitung verwendet werden:

Icon	Name	Index	Bewertung	Kommentar
<input checked="" type="checkbox"/>	Drucker	9	ausreichend	Zu erkennen, aber altbacken
<input checked="" type="checkbox"/>	Sand	6	mangelhaft	Würde ich meine Kinder nicht drin spielen lassen
<input checked="" type="checkbox"/>	Windows	21	ausreichend	Sieht irgendwie komisch aus
<input type="checkbox"/>	A	1	befriedigend	Sieht ganz anständig aus
<input type="checkbox"/>	Weg	14	mangelhaft	Seltsamer Weg
<input type="checkbox"/>	Knopf 2	18	befriedigend	Noch ein Button
<input type="checkbox"/>	Eimer	3	ungenügend	Nicht mal Mülltonnen-Oskar würde dort rein
<input type="checkbox"/>	Müll	5	mangelhaft	Nicht erkennbar
<input checked="" type="checkbox"/>	Editor	16	ausreichend	Na ja, gerade noch akzeptabel
<input type="checkbox"/>	Profan	19	befriedigend	Könnte besser aussehen
<input type="checkbox"/>	Gesicht	4	ungenügend	Komisches Gesicht
<input checked="" type="checkbox"/>	DOS	2	gut	Klar verständlich
<input type="checkbox"/>	Baum	8	mangelhaft	Hä? Wo denn?
<input type="checkbox"/>	Wacker	7	mangelhaft	Frisch aus der Kläranlage

Spalten sind horizontal hintereinander gesetzt. Die Listview.dll verwaltet bis zu 64 Spalten pro Listview-Control. Über jeder Spalte befindet sich ein Spaltenbutton. Drückt der Anwender diesen Knopf, dann startet die automatische Sortierung der Listviews (sofern eingeschaltet) nach den Kriterien dieser Spalte. Ein kleiner Pfeil markiert die zuletzt sortierte Spalte und die Sortier-Richtung.

Zeilen sind vertikal untereinander gesetzt, die Anzahl verfügbarer Zeilen eines Listviews richtet sich nur nach dem vorhandenen Speicher und dem verwendeten Betriebssystem. Einige Funktionen der Listview.dll können aber nur mit bis zu maximal 65536 Zeilen umgehen, was aber mehr als ausreichend sein sollte.

Ein Item ist ein einzelner Eintrag in einer Listbox. Itemtexte dürfen nicht länger als 256 Zeichen sein. Sie können linksbündig, zentriert oder rechtsbündig sein.

Selektierte Zeilen sind ausgewählte Zeilen. Eine Selektierung kann in der ersten oder letzten Spalte eines Listviews mittels der Maus ausgeführt werden oder mit Hilfe der Strg-Taste.

Ein Listview kann auch Icons enthalten, ab V1.5 auch in Sub-Spalten!

Auch Checkboxes sind möglich. Der Benutzer kann sie markieren oder wieder entmarkieren. Der Programmierer kann jederzeit den Status aller Checkboxes abfragen oder auch bestimmen. Auch bei den Checkboxes gilt, entweder alle, oder keine. Rein theoretisch sind auch Checkboxes möglich, die nur in bestimmten Zeilen vorhanden sind. Hierbei mogelt Windows aber, denn wenn man mit der Maus dorthin klickt, wo eine Checkbox sein müsste, erscheint diese durch den Mausklick plötzlich... Darum habe ich auf dieses Feature verzichtet (mit Sendmessage aber möglich...!)

Die Listview.dll unterstützt ab Version 1.3 beliebig viele Listviews innerhalb eines Programms, aber nicht mehr als 63 gleichzeitig. Mehr hierzu unter [EraseListview\(\)](#).

Mit den Editfeldern (Edits) kann der Benutzer Itemtexte editieren. Natürlich kann der Programmierer bestimmen, ob ein Listview-Control Edits erlaubt oder nicht.

Editfelder sind in allen Spalten möglich, mit [SelectColumnEdits\(\)](#) kann aber bestimmt werden, dass nur ausgewählte Spalten editiert werden dürfen.

Mit den Cursortasten kann Spalte oder Zeile verändert werden, in der gerade editiert wird.

Mit der Taste "Bild hoch" springt das Edit an den oberen Rand des Listviews, mit "Bild runter" an den unteren Rand.

"Pos 1" springt ganz nach links und "Ende" ganz an den rechten Rand.

"Tab" springt eins nach rechts, "Tab + Shift" springt eins nach links. Alles in allem also sehr komfortabel!

Drag&Drop ist seit Version 1.7 dabei. Das einfache Verschieben von Texten per Maus lässt Programme leicht bedienbar machen.

## **Was es für den Programmierer zu beachten gilt:**

1. Die User-ID eines Listviews (GWL\_ID) darf nicht verändert werden, z.B. mit SetWindowLong(). Wird sie doch verändert, führt das dazu, dass das Listview im Subclassing nicht gefunden wird. Die DLL nutzt diese Methode, um an einer kritischen Stelle (Windows 98/ME) sicherzustellen, dass wirklich ein passendes Listview vorliegt. Aus einem ähnlichen Grund darf auch GWL\_USERDATA nicht verändert werden, da die DLL diesen Wert benutzt, z.B. für EnableEdits() und EnableDragDrop().

Auch GWL\_USERDATA des Stammfensters, in dem sich Listviews befinden, ist absolut tabu.

2. Die Listview.dll ist optimiert für Windows XP/2000 und aktueller. Natürlich funktioniert sehr vieles auch noch auf älteren Systemen wie 95/98/ME, aber vielleicht nicht alles. Beim Editieren in Listviews ([EnableEdits\(\)](#)) kann es dann u.U. zu Problemen kommen, sodass diese Option nur für neuere Systeme zugelassen werden sollte. Weitere Inkompatibilitäten sind derzeit aber nicht bekannt.

## **Listview.dll und Usermessages:**

Seit Version 1.9 verschickt die Listview.dll verschiedene Usermessages an das Hauptprogramm, wenn bestimmte Ereignisse passieren. An dieser Stelle werden diese Messages genannt und erklärt (in logischer, aber nicht chronologischer Reihenfolge):

**\$1400:** Ermittelt jeden Tastendruck in einem beliebigen Listview.

uwParam = Listview-Handle

ulParam = Tastencode

**\$1401:** Linke Maustaste wurde auf einem Spaltenbutton gedrückt.

uwParam = Listview-Handle

ulParam = Spaltenindex (nullbasierend)

**\$1404:** Rechte Maustaste wurde auf einem Spaltenbutton gedrückt.

uwParam = Listview-Handle

ulParam = Spaltenindex (nullbasierend)

**\$1402:** Anwender will gerade Spaltenbreite verändern.

uwParam = Listview-Handle

ulParam = Spaltenindex (nullbasierend)

**\$1403:** Anwender hat Spaltenbreite verändert.

uwParam = Listview-Handle

ulParam = Spaltenindex (nullbasierend)

**\$1405:** Linke Maustaste wurde in einem Listview geklickt.

uwParam = Spaltenindex (nullbasierend)

ulParam = Zeilenindex (nullbasierend)

Mit GetVar(7) kann das Listview-Handle ermittelt werden, in dem der Klick auftrat.

**\$1406:** Rechte Maustaste wurde in einem Listview geklickt.

uwParam = Spaltenindex (nullbasierend)

ulParam = Zeilenindex (nullbasierend)

Mit GetVar(7) kann das Listview-Handle ermittelt werden, in dem der Klick auftrat.

**\$1407:** Linke Maustaste wurde in einem Listview doppelt geklickt.

uwParam = Spaltenindex (nullbasierend)

ulParam = Zeilenindex (nullbasierend)

Mit GetVar(7) kann das Listview-Handle ermittelt werden, in dem der Doppelklick auftrat.

\$1408: Rechte Maustaste wurde in einem Listview doppelt geklickt.

uwParam = Spaltenindex (nullbasierend)

ulParam = Zeilenindex (nullbasierend)

Mit GetVar(7) kann das Listview-Handle ermittelt werden, in dem der Doppelklick auftrat.

Bitte beachten: Usermessage \$1408 funktioniert nicht, wenn die automatische Editierung (mittels Rechtsklick) eingeschaltet ist, weil der zweite Rechtsklick schon ins Editcontrol geht.

Beispiel (beinhaltet nicht alle Usermessages):

```
Usermessages $1400,$1401,$1402,$1403,$1404

While 1

    WaitInput
    Case %key=2:BREAK

    If %umessage=$1400
        Print "Taste mit Scancode "+Str$(&ulParam)+" gedrückt in Listview "+Str$
(&uwParam)

    ElseIf %umessage=$1401
        Print "Linksklick auf Spaltenknopf in Spalte "+Str$(&ulParam)+" Listview "+Str$
(&uwParam)

    ElseIf %umessage=$1404
        Print "Rechtsklick auf Spaltenknopf in Spalte "+Str$(&ulParam)+" Listview
"+Str$(&uwParam)

    ElseIf %umessage=$1402
        Print "Spaltenbreite wird verändert in Spalte "+Str$(&ulParam)+" Listview
"+Str$(&uwParam)

    ElseIf %umessage=$1403
        Print "Spaltenbreite wurde verändert in Spalte "+Str$(&ulParam)+" Listview
"+Str$(&uwParam)

    EndIf

EndWhile
Usermessages 0
```

## Hinweise für andere Programmsprachen:

Alle Beispiele in dieser Anleitung sind in der Programmsprache "Profan<sup>2</sup>" geschrieben.

Hier eine kurze (unvollständige) Beschreibung des Profan-Autors:

- Eine einfache - an BASIC angelehnte - Syntax, auch - aber nicht nur - für den Anfänger
- Traditionelle prozedurale, aber auch objektorientierte Programmierung. Ganz nach Wahl
- Alle Grafik- und Multimediamöglichkeiten, die Windows bietet
- Umfangreiche Datei- und Verwaltungsfunktionen
- Das komplette Programm in einer nicht zu großen Datei

Weitere Infos gibt es auf <http://www.xprofan.de>

Wenn irgendwo in den Funktionsbeschreibungen von Bereichen (bereich#) geschrieben wird, so ist ein ordinärer Speicherbereich gemeint, wie er z.B. mit der API GlobalAlloc() erzeugt werden kann. Als Parameter muß dann ein Zeiger auf den Speicherbereich übergeben werden.

Strings müssen immer mit einem Nullbyte abschliessen. Bitte beachten, das viele Funktionen einen Zeiger auf einen String erwarten, nicht den String selber!

Alle sonstigen übergebenen Parameter müssen Long-Integer (Longword) Zahlen sein. Diese decken alle Zahlen ab von -2.147.483.647 bis 2.147.483.647 (ca. -2 bis 2 Milliarden). Auch die Rückgabe-Ergebnisse von Funktionen sind Long-Integer.

### **CSV-Dateien:**

Die Listview.dll ist imstande, ganze CSV-Dateien in ein Listview zu importieren und zu exportieren. Aber was ist "CSV"?

"CSV" bedeutet "Comma Separated Values" und ist eine Datei deren Felder durch Kommas (manchmal auch Semikolons) getrennt sind. Es handelt sich dabei um ein textbasierendes Dateiformat, das oft zur Datensicherung von Datenbanken und Tabellenkalkulationen benutzt wird. Quasi jedes bekannte Programm, das irgendwie mit Tabellen zu tun hat, kann dieses Format laden.

Beispielhaft könnte eine CSV Datei so aussehen:

```
Das ist der Titel,1234,Das ist die Beschreibung
```

oder so...

```
Das ist der Titel;1234;Das ist die Beschreibung
```

oder so...

```
"Das ist der Titel",1234,"Das ist die Beschreibung"
```

oder so...

```
"Das ist der Titel";"1234";"Das ist die Beschreibung"
```

Der Listview.dll ist egal, ob als Trennzeichen Komma oder Semikolon verwendet wird (weitere Trennzeichen sind mit [ExchangeSeparator\(\)](#) möglich).

Auch Felder in Anführungszeichen sind erlaubt - wenn in einem Feld beispielsweise Kommas oder Semikolons benutzt werden, z.B. 1246,35.

Leerzeilen in CSV-Dateien werden ebenfalls heraus gefiltert.

Was nicht erlaubt ist, sind Anführungszeichen innerhalb von Anführungszeichen:

```
"Das ist "der" Titel";"1234";"Das ist "die" Beschreibung"
```

Ok, öffnet man eine CSV-Datei per Importfunktion in einer Tabellenkalkulation, werden daraus drei Felder:

FELD1 -> Das ist der Titel

FELD2 -> 1234

FELD3 -> Das ist die Beschreibung

CSV Dateien können auf verschiedene Weise erstellt werden. Zumeist werden sie mit Hilfe eines Tabellenkalkulationsprogramms wie Microsoft Excel angelegt und dann als CSV gespeichert. (Im Datei "Menü" von Microsoft Excel den Menüpunkt "Speichern unter" auswählen -> dann als Dateiformat CSV (Trennzeichen-Getrennt) wählen und die Datei speichern).

Der Vorteil von CSV liegt klar auf der Hand, es ist weit verbreitet, kann leicht erstellt werden und ermöglicht auf

einfache und schnelle Weise speicherbare Listviews.

## **Funktionen:**

### **Funktionen der Listview.dll**

(logisch geordnet nach Themen.)

#### **Dateien:**

[ReadFileQuick](#)  
[WriteFileQuick](#)

#### **Dateiformate:**

[CsvToHeader](#) *umbenannt!*  
[CsvToListview](#)  
[DbfToCsv](#)  
[ExchangeSeparator](#)  
[FilelistToCsv](#) *erweitert!*  
[HeaderToCsv](#) *neu!*  
[ListviewToCsv](#)  
[ListviewToDbf](#)  
[ListviewToRaw](#)  
[SwapLines](#)  
[RawToListview](#)

#### **Grafiken:**

[ArelconsPresent](#) *erweitert!*  
[CreateImageList](#)  
[GetIcon](#)  
[SetBackImage](#)  
[SetIconColumn](#)  
[SetIcon](#)  
[SetIconMode](#)  
[SetIconsFromMem](#)  
[SetIconsWith](#)  
[SetImageList](#)

#### **Checkboxes:**

[AreCheckboxesPresent](#)  
[CheckIfMarked](#)  
[GetAllCheckboxStates](#)  
[GetCheckboxState](#)  
[GetChecked](#)  
[MarkIfChecked](#)  
[SetAllCheckboxStates](#)  
[SetCheckboxState](#)

#### **Texte:**

### **Funktionen der Listview.dll**

(in alphabetischer Reihenfolge)

[AddItemValues](#)  
[AreCheckboxesPresent](#)  
[ArelconsPresent](#)  
[ASortListview](#)  
[BuildListview](#)  
[CheckIfMarked](#)  
[ClearListview](#)  
[CloseMessages](#)  
[ConvertDatas](#)  
[CopyCounnTo](#)  
[CopyLineTo](#)  
[CreateImageList](#)  
[CreateListview](#)  
[CryptMem](#)  
[CsvToHeader](#)  
[CsvToListview](#)  
[DbfToCsv](#)  
[DeleteAllItems](#)  
[DeleteColumn](#)  
[DeleteDoubleItems](#)  
[DeleteItem](#)  
[DeleteSpaceLines](#)  
[EditManual](#)  
[EnableDragDrop](#)  
[EnableEdits](#)  
[EraseListview](#)  
[ExamineColumn](#)  
[ExchangeBytes](#)  
[ExchangeSeparator](#)  
[FilelistToCsv](#)  
[ForbidScrollMessage](#)  
[GetAllCheckboxStates](#)  
[GetAllSelected](#)  
[GetCheckboxState](#)  
[GetChecked](#)  
[GetColumnName](#)  
[GetColumns](#)  
[GetColumnUpdate](#)  
[GetColumnWidth](#)  
[GetControlParas](#)  
[GetDllVersion](#)  
[GetDragDropParas](#)  
[GetEdgeFloats](#)  
[GetEdgeIntegers](#)  
[GetFloat](#)  
[GetIcon](#)  
[GetIndex](#)  
[GetItemState](#)  
[GetItemText](#)  
[GetItemTextEx](#)  
[GetItemTextsAsFloat](#)

[ExamineColumn](#)  
[GetItemText](#)  
[GetItemTextEx](#)  
[GetItemTextsAsFloat](#)  
[GetItemTextsAsInteger](#)  
[GetLineText](#)  
[SearchBlankItem](#)  
[SearchText](#)  
[SetItemText](#)  
[SetItemTextEx](#)  
[SetLineNumbers](#)

*neu!*

#### **Zahlen:**

[AddItemValues](#)  
[ConvertDatas](#)  
[ExamineColumn](#)  
[ExchangeBytes](#)  
[GetEdgeFloats](#)  
[GetEdgeIntegers](#)  
[GetFloat](#)  
[SetLineNumbers](#)

#### **Sortierung:**

[ASortListview](#)  
[SortManual](#)

#### **Drucken:**

[PrintColumns](#)  
[PrintListview](#)  
[SetPrintAttributes](#)

#### **Events:**

[GetLastKey](#)  
[GetSelected](#)  
[GetSelectedDbClk](#)  
[GetVar](#) *erweitert!*  
[User-Messages](#) *erweitert!*

#### **Editieren:**

[EditManual](#)  
[EnableEdits](#)  
[GetVar](#)  
[SelectColumnEdits](#)

#### **Drag & Drop:**

[EnableDragDrop](#) *erweitert!*  
[GetDragDropParas](#) *erweitert!*

#### **Markierung:**

[GetItemTextsAsInteger](#)  
[GetLastKey](#)  
[GetLines](#)  
[GetLineText](#)  
[GetNeededMemory](#)  
[GetNullOffset](#)  
[GetOwnControlParas](#)  
[GetRealColumnIndex](#)  
[GetSelected](#)  
[GetSelectedCount](#)  
[GetSelectedDbClk](#)  
[GetSelectedLine](#)  
[GetTabOffsets](#)  
[GetVar](#)  
[GetVisibleColumns](#)  
[GetVisibleLines](#)  
[HeaderToCsv](#)  
[IColumn](#)  
[InitMessages](#)  
[ListviewToCsv](#)  
[ListviewToDbf](#)  
[ListviewToRaw](#)  
[MarkIfChecked](#)  
[MixRGBs](#)  
[PrintColumns](#)  
[PrintListview](#)  
[RaiseColumns](#)  
[RaiseLine](#)  
[RawToListview](#)  
[ReadFileQuick](#)  
[SearchBlankItem](#)  
[SearchText](#)  
[SelectColumnEdits](#)  
[SelectLine](#)  
[SetAllCheckboxStates](#)  
[SetBackImage](#)  
[SetCheckboxState](#)  
[SetColumnAlignment](#)  
[SetColumnName](#)  
[SetColumnSort](#)  
[SetColumnWidth](#)  
[SetColumnsWidthLimits](#)  
[SetColumnUpdate](#)  
[SetFilelistFilter](#)  
[SetFilelistNoFilter](#)  
[SetIconColumn](#)  
[SetIcon](#)  
[SetIconMode](#)  
[SetIconsFromMem](#)  
[SetIconsWith](#)  
[SetImageList](#)  
[SetIndex](#)  
[SetItemText](#)  
[SetItemTextEx](#)  
[SetLineHeight](#)  
[SetLineNumbers](#)  
[SetListviewStyle](#)  
[SetPrintAttributes](#)  
[ShowListview](#)  
[SItem](#)  
[SortManual](#)  
[SwapLines](#)  
[UpdateListview](#)  
[WriteFileQuick](#)

[GetAllSelected](#)  
[GetSelectedCount](#) *neu!*  
[GetSelectedLine](#)  
[GetItemState](#) *neu!*  
[SelectLine](#)

#### **Daten löschen:**

[ClearListview](#)  
[DeleteAllItems](#) *neu!*  
[DeleteColumn](#) *neu!*  
[DeleteDoubleItems](#)  
[DeleteItem](#) *neu!*  
[DeleteSpaceLines](#)

#### **Listview-Struktur:**

[BuildListview](#)  
[ClearListview](#)  
[CreateListview](#)  
[EraseListview](#)  
[Icolumn](#)  
[Sitem](#)

#### **Get/Set-Funktionen:**

[GetColumnName](#)  
[GetColumns](#)  
[GetColumnUpdate](#)  
[GetColumnWidth](#)  
[GetControlParas](#)  
[GetDllVersion](#)  
[GetIndex](#)  
[GetLines](#)  
[GetNeededMemory](#)  
[GetNullOffset](#)  
[GetOwnControlParas](#)  
[GetRealColumnIndex](#)  
[GetTabOffsets](#)  
[GetVar](#) *erweitert!*  
[GetVisibleColumns](#) *neu!*  
[GetVisibleLines](#) *neu!*  
[SetColumnAlignment](#)  
[SetColumnName](#)  
[SetColumnSort](#)  
[SetColumnWidth](#) *neu!*  
[SetColumnsWidthLimits](#)  
[SetColumnUpdate](#)  
[SetFilelistFilter](#)  
[SetFilelistNoFilter](#)  
[SetIndex](#)  
[SetLineHeight](#)  
[SetListviewStyle](#) *umbenannt!*

#### **Sonstiges:**

[CloseMessages](#)  
[CopyColumnTo](#)  
[CopyLineTo](#)  
[CryptMem](#)

#### **Funktionen der Listview.dll**

(Kompakt ABC)

[AddItemValues](#) [AreCheckboxesPresent](#) [ArelconsPresent](#)  
[ASortListview](#) [BuildListview](#) [CheckIfMarked](#) [ClearListview](#)  
[CloseMessages](#) [ConvertDatas](#) [CopyColumnTo](#)  
[CopyLineTo](#) [CreateImageList](#) [CreateListview](#) [CryptMem](#)  
[CsvToHeader](#) [CsvToListview](#) [DbfToCsv](#) [DeleteAllItems](#)  
[DeleteColumn](#) [DeleteDoubleItems](#) [DeleteItem](#)  
[DeleteSpaceLines](#) [EditManual](#) [EnableDragDrop](#)  
[EnableEdits](#) [EraseListview](#) [ExamineColumn](#)  
[ExchangeBytes](#) [ExchangeSeparator](#) [FilelistToCsv](#)  
[ForbidScrollMessage](#) [GetAllCheckboxStates](#)  
[GetAllSelected](#) [GetCheckboxState](#) [GetChecked](#)  
[GetColumnName](#) [GetColumns](#) [GetColumnUpdate](#)  
[GetColumnWidth](#) [GetControlParas](#) [GetDllVersion](#)  
[GetDragDropParas](#) [GetEdgeFloats](#) [GetEdgeIntegers](#)  
[GetFloat](#) [GetIcon](#) [GetIndex](#) [GetItemState](#) [GetItemText](#)  
[GetItemTextEx](#) [GetItemTextsAsFloat](#)  
[GetItemTextsAsInteger](#) [GetLastKey](#) [GetLines](#)  
[GetLineText](#) [GetNeededMemory](#) [GetNullOffset](#)  
[GetOwnControlParas](#) [GetRealColumnIndex](#) [GetSelected](#)  
[GetSelectedCount](#) [GetSelectedDbClk](#) [GetSelectedLine](#)  
[GetTabOffsets](#) [GetVar](#) [GetVisibleColumns](#)  
[GetVisibleLines](#) [HeaderToCsv](#) [Icolumn](#) [InitMessages](#)  
[ListviewToCsv](#) [ListviewToDbf](#) [ListviewToRaw](#)  
[MarkIfChecked](#) [MixRGBs](#) [PrintColumns](#) [PrintListview](#)  
[RaiseColumns](#) [RaiseLine](#) [RawToListview](#) [ReadFileQuick](#)  
[SearchBlankItem](#) [SearchText](#) [SelectColumnEdits](#)  
[SelectLine](#) [SetAllCheckboxStates](#) [SetBackImage](#)  
[SetCheckboxState](#) [SetColumnAlignment](#) [SetColumnName](#)  
[SetColumnSort](#) [SetColumnWidth](#) [SetColumnsWidthLimits](#)  
[SetColumnUpdate](#) [SetFilelistFilter](#) [SetFilelistNoFilter](#)  
[SetIconColumn](#) [SetIcon](#) [SetIconMode](#)  
[SetIconsFromMem](#) [SetIconsWith](#) [SetImageList](#) [SetIndex](#)  
[SetItemText](#) [SetItemTextEx](#) [SetLineHeight](#)  
[SetLineNumbers](#) [SetListviewStyle](#) [SetPrintAttributes](#)  
[ShowListview](#) [Sitem](#) [SortManual](#) [SwapLines](#)  
[UpdateListview](#) [WriteFileQuick](#)



[ForbidScrollMessage](#)

[InitMessages](#)

[MixRGBs](#)

[RaiseColumns](#)

[RaiseLine](#)

[ShowListview](#)

[UpdateListview](#)

*umbenannt!*

### **CreateListview(F,I,T,H,G,S)**

Erzeugt ein leeres Listview-Control. Es bleibt aber noch solange unsichtbar, bis [ShowListview\(\)](#) aufgerufen wurde.

F : Long - Handle des Fensters, in dem das Listview erstellt werden soll (z.B.: %HWND).

I : Long - Instance Handle des Fensters F (z.B.%HINSTANCE).

T : Long - Textfarbe für Itemtexte (oder -1 für default)

H : Long - Hintergrundfarbe des Listviews (oder -1 für default)

G : Long - Texthintergrundfarbe für Itemtexte (oder -1 für default)

S : Long - Styles für das Listview (LVS\_EX\_STYLES)

Rückgabe-Ergebnis: Long - Handle des Listviews. Bei Fehler = 0.

Das Listview, das hier erzeugt wird, ist noch leer und unsichtbar.

T, H und G sind RGB-Werte. Wird bei T und H eine -1 angegeben, dann wird die vom System voreingestellte Farbe verwendet. Wird bei G eine -1 angegeben, dann wird ein transparenter Hintergrund gewählt.

Hier einige interessante Styles für S (die Werte können addiert werden):

\$1 = Gitter Hilfslinien einschalten (Grids)

\$2 = Subicons verwenden (Icons dürfen in allen Spalten erscheinen, nicht nur in der linken)

\$4 = Checkboxes einschalten (jede Zeile erhält eine (abfragbare) Checkbox)

\$10 = Spaltenbuttons per Drag&Drop verschiebbar machen

\$20 = Selektionen erstrecken sich über die ganze Zeile, nicht nur bis zur zweiten Spalte

\$400 = Der Anwender kann die Spaltenbreite nicht verändern

Ich empfehle den Style \$20 immer mit zu setzen. Um mehr Informationen zu erhalten, empfiehlt es sich in der Win32 Programmer's Reference unter LVS\_EX\_STYLES nachzulesen. Alle Styles sind in Listview-Controls im Report-View-Modus aber nicht möglich.

Die Aktivierung und Auswertung von Checkboxes (Style \$4) werden von der Listview.dll ab Version 1.1 voll unterstützt.

```
listview&=CreateListview(%hwnd,%hinstance,0,Rgb(255,255,255),-1,$31)
```

Um ein Listview wieder zu beenden/zerstören, empfiehlt sich folgender Weg:

```
EraseListview(listview&) 'Listviewstrukturen entfernen
DestroyWindow(listview&) 'Profan Befehl, um Fensterobjekte zu zerstören
CloseMessages(%hwnd) 'Subclassing des Parentfensters entfernen
```

### **IColumn(H,T,B,F)**

### **InsertColumn H,T,B,F**

Generiert rechts in ein Listview eine neue Spalte.

IColumn() ist die eigentliche DLL-Funktion, für Profan existiert auch die Definition InsertColumn.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

T : Long - (Adresse auf einen) Text des Spaltenbuttons.

B : Long - Breite der Spalte in Pixel  
F : Long - Textformatierungs-Flag

Rechts im Listviews wird eine neue Spalte erzeugt und die angegebenen Texte werden als neue Items eingefügt.  
Um die neue Spalte an anderer Stelle einzufügen, ist [SetIndex\(\)](#) zu benutzen.  
T ist der Text, den der Spaltenbutton erhält. Der Text sollte nicht länger sein als 256 Buchstaben.  
B ist die Breite, gemessen in Bildschirm Pixeln  
Mit F kann die Textausrichtung aller Items in dieser Spalte bestimmt werden. Flags für F können sein:

0 = linksbündig  
1 = rechtsbündig  
2 = zentriert

Die linke Spalte (Index 0) kann nur linksbündigen Text beinhalten. Mit einem Trick kann das Listview aber überrumpelt werden, indem die Textausrichtung nachträglich geändert wird. Das kann mit [SetColumnAlignment\(\)](#) erreicht werden.

#### Beispiel für die Erzeugung einer neuen Spalte:

```
text$="Neue Spalte"  
IColumn(listview&,addr(text$),120,0)
```

oder

```
InsertColumn listview&,"Neue Spalte",120,0
```

#### **SItem(H,B,A)**

#### **SetItem H,S1,S2,S3,...,S12**

Erzeugt eine neue Zeile und fügt Items in die neuen Zeile ein.  
SItem() ist die eigentliche DLL-Funktion, für Profan existiert auch die Definition SetItem.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
B : Zeiger auf einen Speicherbereich mit Stringadressen.  
A : Long - Anzahl Strings, die in Bereich B verwendet werden.

Am Ende des Listviews (unten) wird eine neue Zeile erzeugt und die angegebenen Texte werden als neue Items eingefügt.

Um die neue Zeile an anderer Stelle einzufügen, ist [SetIndex\(\)](#) zu benutzen.

B ist ein Speicherbereich, in dem Zeiger auf Strings als LongInt gespeichert wurden, A gibt an, wieviele Strings übergeben werden sollen.

Maximal werden bis zu 64 Items pro Zeile von der Listview.dll unterstützt.. Wenn sie die Profan-Definition SetItem benutzen, können damit maximal 12 Items übergeben werden, mehr Parameter unterstützt Profan nicht. Mit Profan 5 sind es sogar nur 9 Items... für normale Listviews reicht's aber.

Itemtexte sollten nicht länger als 256 Zeichen sein!

#### Beispiel für 63 Items:

```
dim bereich#,16384  
  
text1$="Neues_Item 1"  
text2$="Neues_Item 2"  
text3$="Neues_Item 3"  
...  
text63$="Neues_Item 63"  
  
long bereich#,0=addr(text1$)
```

```

long bereich#,4=addr(text2$)
long bereich#,8=addr(text3$)
...
long bereich#,252=addr(text63$)

SItem(listview&,bereich#,63)

Dispose bereich#

```

oder (hier mit 12 Items)

```
SetItem listview&,"Neues_Item 1","Neues_Item 2","Neues_Item 3",...,"Neues_Item 12"
```

### **ShowListview(H,X,Y,B,H)**

Zeigt ein Listview auf dem Bildschirm an, das mit CreateListview() erzeugt wurde.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
X : Long - X-Koordinate des Listviews  
Y : Long - Y-Koordinate des Listviews  
B : Long - Breite des Listviews  
H : Long - Höhe des Listviews

Das Listview mit dem Handle H wird jetzt sichtbar gemacht, an Position X, Y und in Größe B x H.

Es gibt keine spezielle Funktion, um ein Listview wieder freizugeben oder zu beenden, dazu kann Profan's [@DestroyWindow\(\)](#) oder die Windows-API DestroyWindow() benutzt werden. Wird das Fenster geschlossen, in dem sich das Listview befindet, dann zerstört Windows automatisch alle Objekte, die sich darin befinden. Bevor ShowListview() aufgerufen wird, sollte das Listview mit Text und/oder Icons gefüllt werden. Vorher nämlich, in unsichtbarem Zustand, erfolgt das Füllen mit Text und Grafik mit 3-4 facher Geschwindigkeit!!!

```
ShowListview(listview&,100,100,600,400)
```

### **InitMessages(F)**

Initiiert das Subclassing für alle verwendeten Listviews im Fenster F.

F : Long - Handle des Fensters, in dem ein Listview erstellt wurde, z.B. (%HWND).

Erst durch diese Funktion wird die automatische Sortierung und die Abfrage angeklickter oder markierter Items möglich.

InitMessages() darf nicht für jedes Listview verwendet werden, muß aber für jedes Fenster aufgerufen werden, in dem mindestens ein Listview verwendet wird.

Mehrfachaufrufe für ein und dasselbe Fenster sind nicht gestattet und führen zu Fehlern!

InitMessages() zapft alle Messages für ein Fenster ab, bevor sie dorthin gelangen. Bestimmte Messages werden dann für eigene Zwecke mißbraucht oder verändert.

Am besten ist es, zuerst alle Listviews aufzubauen und mit Text zu füllen, und zum Schluss für jedes Fenster, in dem sich (mindestens) ein Listview befindet, InitMessages() auszuführen.

Beendet wird die Funktion mit [CloseMessages\(\)](#), am besten, kurz bevor die Listview.dll wieder entladen wird.

```
InitMessages(%HWND)
InitMessages(DIALOGFENSTER%) .
```

### **CloseMessages(F)**

Deaktiviert das Subclassing für alle verwendeten Listviews im Fenster F wieder.

F : Long - Handle des Fensters, in dem ein ListView erstellt wurde, z.B. (%HWND).

CloseMessages() wird am besten aufgerufen, kurz bevor die ListView.dll wieder entladen wird.

Alle Fenster, die mit [InitMessages\(\)](#) subclassed wurden, müssen auch wieder deaktiviert werden, und zwar auf jeden Fall, bevor das jeweilige Fenster geschlossen wird!

```
CloseMessages(%HWND)
```

und/oder

```
CloseMessages(DIALOGFENSTER%) .
```

Wenn der Anwender im ListView editiert hat, gibt Windows beim Beenden mit CloseMessages() einen Warnton aus.

Das ist deshalb so, weil seit Version 1.5 auch die Editfelder im ListView-Control subclassed werden.

Der Warnton kann verhindert werden, wenn das ListView vor dem CloseMessages() geschlossen wird, was mit DestroyWindow() möglich ist.

Mit Erscheinen von XProfan änderten sich einige interne Profan-Sachen, sodass ich grundsätzlich empfehle, das Parentfenster eines Listviews zum Programmende mit DestroyWindow() zu schliessen. CloseMessages() ist dann oft gar nicht mehr nötig, löst teilweise sogar Fehler hervor, weil XProfan 8/9/10 mehrmals verändert wurde. Hier muss der ListView.dll Programmierer ein wenig probieren, sollte es in dieser Beziehung mal Probleme geben, je nach Profanversion und Betriebssystem.

```
DestroyWindow(listview&)    'und/oder  
CloseMessages(%hwnd)
```

### **ASortListView(H,B,A)**

#### **AutoSortListView H,S1,S2,S3,...,S12**

Bestimmt, in welcher Art und Weise jede einzelne Spalte eines Listviews sortiert werden soll, sobald der Anwender den entsprechenden Spaltenbutton gedrückt hat.

ASortListView() ist die eigentliche DLL-Funktion, für Profan existiert auch die Definition AutoSortListView.

H : Long - Handle eines mit [CreateListView\(\)](#) erstellten ListView Controls.

B : Zeiger auf einen Speicherbereich mit Longint-Werten.

A : Long - Anzahl Longint's, die in Bereich B verwendet werden.

Die Listviews der ListView.dll bieten eine automatische Sortierung, sobald der Anwender einen Spaltenbutton drückt. Voreingestellt ist "keine Sortierung", mit dieser Funktion aber kann bestimmt werden, ob eine Sortierung nach dem Alphabet oder eine Sortierung nach Zahlenwerten erfolgen soll.

Die Longint's in B geben hierzu in der Reihenfolge der Spalten an, welches Flag für jede Spalte gesetzt wird:

```
0 = Nicht automatisch sortieren  
1 = Sortieren nach Alphabet  
2 = Sortieren nach Zahlenwert
```

Wenn ein ListView also z.B. 5 Spalten besitzt, ist es logisch, für A eine 5 anzugeben und in B 5 Longint's zu schreiben...

Das folgende Beispiel würde also festlegen, das die beiden ersten Spalten nach Alphabet sortiert werden, die dritte Spalten nicht automatisch sortiert wird und die 4te und 5te Spalte nach Zahlenwerten sortiert werden.

Beispiel für 5 Items:

```
dim bereich#,20
```

```

long bereich#,0=1
long bereich#,4=1
long bereich#,8=0
long bereich#,12=2
long bereich#,16=2

ASortListview(listview&,bereich#,5)

Dispose bereich#

```

### oder als Definition

```
AutoSortListview listview&,1,1,0,2,2
```

Maximal werden bis zu 64 Spalten pro Listview von der Listview.dll unterstützt.. Wenn sie die Profan-Definition AutoSortListview benutzen, können damit maximal 14 LongInt's übergeben werden, mehr Parameter unterstützt Profan nicht. Mit Profan 5 sind es sogar nur 11 Longint's... für normale Listviews reicht's aber.

Ab V1.5 wurde die Zahlensortierung erheblich erweitert. So werden jetzt auch negative und/oder dezimale Zahlen richtig sortiert. Ob für dezimale Zahlen ein Punkt oder ein Komma als Trennzeichen verwendet wird, ist egal. Die Sortierung ist dadurch nur unmerklich langsamer geworden (ich könnte sie noch erheblich steigern, nur würde die Dll dann nur noch ab einem .686 Prozessor funktionieren, wollt ihr das?).

Alle Zahlenspalten werden jetzt als 64 Bit Fließkommazahlen gelesen, wobei folgende Zeichen erlaubt sind:

- 0 1 2 3 4 5 6 7 8 9 , .

### **ReadFileQuick(S,B,O,A)**

Schnelles Lesen einer CSV-Datei in einem Bereich.

S : Zeiger auf einen Strings mit Namen einer Datei.  
 B : Long - Bereich#, in den die Daten der Datei gelesen werden.  
 O : Long - Offset im Bereich#.  
 A : Long - Anzahl Bytes, die gelesen werden sollen.

Ergebnis: Long - Anzahl an Bytes, die gelesen wurden, bei Fehler 0.

Schneller und bequemer kann man eine Datei wohl nicht mehr laden. ReadFileQuick() ist dafür gedacht, [CSV-Dateien](#) schnell in den Speicher zu laden.

Es kann aber für jede beliebige Datei benutzt werden. Wer die ProSpeed.dll kennt, ReadFileQuick() ist gleich ReadFileFast()...

Aus der Datei S werden A Bytes in den Bereich B (plus Offset O) geladen. Es wird im Modus "nur lesen" geladen. Beispiel:

```

text$="Datei.csv"
bytes=&@FileSize(text$)
If bytes>0
  Dim bereich#,bytes&
  ReadFileQuick(addr(text$),bereich#,0,bytes&)
  ...
  Dispose bereich#
EndIf

```

### **WriteFileQuick(S,B,O,A)**

Schnelles Schreiben eines Bereichs in eine CSV-Datei.

S : Zeiger auf einen Strings mit Namen einer Datei.  
B : Long - Bereich#, aus dem die Daten in eine Datei geschrieben werden sollen.  
O : Long - Offset im Bereich#.  
A : Long - Anzahl Bytes, die geschrieben werden sollen.

Ergebnis: Long - Anzahl an Bytes, die geschrieben wurden, bei Fehler 0.

Schneller und bequemer kann man eine Datei wohl nicht mehr speichern. WriteFileQuick() ist dafür gedacht, [CSV-Dateien](#) im Speicher schnell auf einen Datenträger zu speichern.

Es kann aber für jede beliebige Datei benutzt werden. Wer die ProSpeed.dll kennt, WriteFileQuick() ist gleich WriteFileFast()...

Aus dem Bereich B werden A Bytes in die Datei S (plus Offset O) geschrieben. Es wird im Modus "nur schreiben" gespeichert. Wenn schon eine Datei des gleichen Names existiert, wird sie überschrieben.

Beispiel:

```
text$="Datei.csv"  
WriteFileQuick(addr(text$),bereich#,0,bytes&)
```

### **SwapLines(B,A,F)**

Tauscht alle Zeilen einer csv.-Datei (oder Textdatei) von oben nach unten.

B : Zeiger auf einen Bereich, in den mit [ReadFileQuick\(\)](#) eine Textdatei geladen wurde.

A : Long - Größe der Datei in Bytes

F : Long - Flag

Ergebnis: Long - Neue Größe der Datei in Bytes, bei Fehler 0.

Hiermit kann die Reihenfolge der Zeilen einer Textdatei geändert werden. Diese Funktion ändert nichts in einem Listview, die Reihenfolge eines Textes kann verändert werden, bevor ein Text in ein Listview eingelesen wird.... Zusätzlich werden in dem Text alle Leerzeilen entfernt. Es werden auch alle Chr\$(10) aus der Datei entfernt werden, sofern F = 0 ist. Ist F = 1, dann bleiben alle Chr\$(10) erhalten.

Beispiel:

```
SwapLines(bereich#,127,1)
```

Aus dieser CSV-Datei...

```
Duisburg-Homberg;02066  
Muelheim a.d.Ruhr;0208  
Oberhausen Rheinl.;0208  
Gelsenkirchen;0209  
Ratingen;02102  
Hilden;02103
```

wird dann diese...

```
Hilden;02103  
Ratingen;02102  
Gelsenkirchen;0209  
Oberhausen Rheinl.;0208  
Muelheim a.d.Ruhr;0208  
Duisburg-Homberg;02066
```

### **CsvToListview(H,B,A,S)**

Einlesen einer CSV-Datei in ein Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen Bereich, in den mit [ReadFileQuick\(\)](#) eine CSV-Datei geladen wurde.  
A : Long - Größe der Datei in Bytes  
S : Long - Anzahl vorhandener Spalten in Listview H.

Schneller und bequemer kann man ein Listview nicht mehr mit Einträgen füllen, diese Technik ist mit Profan natürlich wesentlich schneller als [SItem\(\)](#).

Die [CSV-Datei](#), die ja Trennzeichen enthält, wird Item für Item dem Listview angehängt, wobei nach jedem Trennzeichen das nächste Item mit Text gefüllt wird.

Neue Zeilen werden automatisch erzeugt. Um die neuen Zeile an anderer Stelle als unten einzufügen, ist [SetIndex\(\)](#) zu benutzen.

S muß gleich der Spaltenaufteilung in der Datei sein...

CsvToListview() verändert die Daten in B, also bitte nicht mehrmals hintereinander mit den gleichen Daten benutzen.

```
CsvToListview(listview&,bereich#,78632,8)
```

Es gibt verschiedene Bytes, die CsvToListview() intern verwendet, und die nicht in B vorkommen sollten. Das sind die ASCII-Codes 0, 2, 3, 4, 254 und 255. Diese Bytes kommen in Csv- oder sonstigen Textdateien normalerweise nie vor.

CsvToListview löscht vor seiner Arbeit alle leeren Zeilen, damit es keine unerwünschten Spalten- bzw. Zeilenvorschübe (bei nur einer Spalte) gibt. ASCII-Code 11 ist ein Sonderfall. Wird dieses gefunden und das Byte zuvor war ein Zeilenvorschub, wann wird eine leere Spalte/Zeile eingefügt. Der Democode "LV\_Quellcode\_einlesen.prf" nutzt diese Technik, um Quellcodes darzustellen zu können, die auch Leerzeilen enthalten. Mehr dazu dort.

### **ListviewToCsv(H,B,A,F)**

Auslesen eines Listviews in eine CSV-Datei.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen Bereich, in den alle Listview-Texte als CSV-Datei gespeichert werden.  
A : Long - ASCII Code des Trennzeichens oder 0 (dann wird ein Komma verwendet)  
F : Long - Flag (0 = Texte in Anführungszeichen setzen / 1 = nur Texte).

Ergebnis: Long - Anzahl Bytes, die in den Bereich B geschrieben wurden.

Eine Funktion, um den Inhalt eines Listviews abzuspeichern. ListviewToCsv() schreibt den Inhalt des Listviews H als [CSV-Datei](#) in den Speicher B. Der Speicher muß natürlich groß genug dimensioniert sein, wie groß er mindestens sein muß, kann vorher mit [GetNeededMemory\(\)](#) ermittelt werden....

Mit [WriteFileQuick\(\)](#) läßt er sich dann auf einem Datenträger abspeichern.

In der CSV-Datei werden Kommas als Trennzeichen verwendet, mit A kann aber auch ein x-beliebiges Trennzeichen ausgewählt werden. Hierzu muß der ASCII-Wert des Zeichens angegeben werden. Hier mal ein paar alternative ASCII-Werte:

```
; = 59  
/ = 47  
- = 45
```

Aber Vorsicht, die Liestview.dll unterstützt nur Kommas oder Semikolons (wie fast alle anderen Programme auch, die CSV-Dateien benutzen).

Es ist aber jederzeit möglich, [ExchangeSeparator\(\)](#) zu verwenden, mit dem jedes beliebige Trennzeichen in einer CSV-Datei verarbeitet werden kann.

Mit F kann bestimmt werden, ob die einzelnen Itemtexte in Anführungszeichen gesetzt werden oder nicht. Das ist dann sinnvoll, wenn Itemtexte Kommas oder Semikolons beinhalten, die ansonsten als Trennzeichen gewertet würden, wenn sie nicht in Anführungszeichen gesetzt sind. Das produziert dann natürlich böse Fehler....

```
ListviewToCsv(listview&,bereich#,0,0)
```

### **GetColumns(H)**

Ermittelt, wie viele Spalten ein Listview besitzt.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Rückgabe-Ergebnis: Long - Anzahl Spalten in Listview H.

### **GetLines(H)**

Ermittelt, wie viele Zeilen ein Listview besitzt.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Rückgabe-Ergebnis: Long - Anzahl Zeilen in Listview H.

### **GetSelected(BT,BL)**

Ermittelt, ob und in welchem Listview zuletzt ein Item angeklickt wurde.

BT : Zeiger auf einen Bereich, der die Texte aller Items in der angeklickten Zeile empfängt.

BL : Zeiger auf einen Bereich, das Handle des angeklickten Listviews empfängt.

Rückgabe-Ergebnis: Long - Anzahl Spalten in Listview H oder 0, wenn kein Linksklick erfolgt ist.

Wieder eine Funktion, die dem User einiges an Arbeit abnimmt.

BT muß ein Speicherbereich sein, der 16384 Bytes groß sein sollte (max. bis zu 63 Spalten a 256 Bytes Text = 16384).

Hierin schreibt GetSelected() alle Item-Texte der ausgewählten Zeile, die einzelnen Texte sind durch Tabs (Chr\$(9)) getrennt, das allerletzte Byte ist ein Nullbyte.

Mit [GetTabOffsets\(\)](#) können die einzelnen Strings später bequem lokalisiert und weiterverarbeitet werden.

BL muß ein 4 Byte großer Speicher sein, hierin schreibt GetSelected() ein LongInt, das Handle des Listviews, das angeklickt wurde.

Jeder Linksklick, der ausgewertet wurde, wird von der Listview.dll abgehakt, sodaß nicht mehrmals auf den gleichen Linksklick abgefragt werden kann / muß.

Hier ein Beispiel, wie eine typische Profan-WaitInput-Schleife aussehen könnte, um vier Listviews abzufragen (Sourcecode ab Profan 7):

( addr(y&) schreibt das Listview-Handle direkt in die Variable y&, mit Profanversionen niedriger als Version 7 muß eine Bereichsvariable verwendet werden, z.B. x&=GetSelected(itemtexts#,bereich#) ... y&=Long(bereich#,0).)

```
While 1
  WaitInput
  Case %key=2:BREAK                                'WaitInput-Schleife verlassen
  x&=GetSelected(itemtexts#,addr(y&))                'Letzten Linksklick auswerten...
```



```

If x&                                'Hat einer stattgefunden? Ja!

Case y&=listview&:Print "Listview 1:"    'Listview ermitteln,
Case y&=listview2&:Print "Listview 2:"    'in dem
Case y&=listview3&:Print "Listview 3:"    'der Linksklick
Case y&=listview4&:Print "Listview 4:"    'ausgelöst wurde

GetTabOffsets(itemtexts#,nurso#)          'Alle Offsets auslesen und alle
Trennzeichen (9)                          'gegen Null austauschen

y&=0
WhileNot x&=y&                        'Schleife zum
    text$=String$(itemtexts#,Long(nurso#,y&*4)) 'Ausgeben aller
    Print text$                          'Item-Texte
    y&=y&+1
EndWhile
EndIf
EndWhile

```

### **GetSelectedDbClk(BT,BL)**

Ermittelt, ob und in welchem Listview zuletzt ein Item per Doppelklick angeklickt wurde.

BT : Zeiger auf einen Bereich, der die Texte aller Items in der angeklickten Zeile empfängt.

BL : Zeiger auf einen Bereich, das Handle des mit Doppelklick angeklickten Listviews empfängt.

Rückgabe-Ergebnis: Long - Anzahl Spalten in Listview H oder 0, wenn kein Doppelklick erfolgt ist.

Noch eine Funktion, die dem User einiges an Arbeit abnimmt.

BT muß ein Speicherbereich sein, der 16384 Bytes groß sein sollte (max. bis zu 64 Spalten a 256 Bytes Text = 16384).

Hierin schreibt GetSelectedDbClk() alle Item-Texte der ausgewählten Zeile, die einzelnen Texte sind durch Tabs (Chr\$(9)) getrennt, das allerletzte Byte ist ein Nullbyte.

Mit [GetTabOffsets\(\)](#) können die einzelnen Strings später bequem lokalisiert und weiterverarbeitet werden.

BL muß ein 4 Byte großer Speicher sein, hierin schreibt GetSelectedDbClk() ein LongInt, das Handle des Listviews, das angeklickt wurde.

Jeder Doppelklick, der ausgewertet wurde, wird von der Listview.dll abgehakt, sodaß nicht mehrmals auf den gleichen Doppelklick abgefragt werden kann / muß.

Hier ein Beispiel, wie eine typische Profan-WaitInput-Schleife aussehen könnte, um vier Listviews abzufragen (Sourcecode ab Profan 7):

( addr(y&) schreibt das Listview-Handle direkt in die Variable y&, mit Profanversionen niedriger als Version 7 muß eine Bereichsvariable verwendet werden, z.B. x&=GetSelectedDbClk(itemtexts#,bereich#) ...

y&=Long(bereich#,0).)

```

While 1
    WaitInput
    Case %key=2:BREAK                                'WaitInput-Schleife verlassen

    x&=GetSelectedDbClk(itemtexts#,addr(y&))          'Letzten Doppelklick auswerten...
    If x&                                              'Hat einer stattgefunden? Ja!

```

```

Case y&=listview&:Print "Listview 1:"      'Listview ermitteln,
Case y&=listview2&:Print "Listview 2:"    'in dem
Case y&=listview3&:Print "Listview 3:"    'der Linksklick
Case y&=listview4&:Print "Listview 4:"    'ausgelöst wurde

GetTabOffsets(itemtexts#,nurso#)          'Alle Offsets auslesen und alle
Trennzeichen (9)                          'gegen Null austauschen

y&=0
WhileNot x&=y&                            'Schleife zum
  text$=String$(itemtexts#,Long(nurso#,y&*4)) 'Ausgeben aller
  Print text$                              'Item-Texte
  y&=y&+1
EndWhile
EndIf
EndWhile

```

### **GetAllSelected(BT,BL)**

Ermittelt, ob und in welchem Listview eine oder mehrere Zeilen markiert wurden (Mehrfachauswahl).

BT : Zeiger auf einen Bereich, der die Texte aller Items in allen markierten Zeilen empfängt.

BL : Zeiger auf einen Bereich, das Handle des angeklickten Listviews empfängt.

Rückgabe-Ergebnis: Long - Anzahl aller markierter Zeilen in Listview H oder 0, wenn keine Markierung vorgenommen wurde.

Noch eine Funktion, die dem User einiges an Arbeit abnimmt.

BT muß ein großer Speicherbereich sein, wie groß er sein muß, erfährt man durch die Funktion

GetNeededMemory().

Hierin schreibt GetAllSelected() alle Item-Texte aller ausgewählter Zeilen, die einzelnen Itemtexte einer Zeile sind durch Tabs (Chr\$(9)) getrennt, jede Zeile wird durch ein Nullbyte von der nächsten Zeile getrennt.

Mit [GetTabOffsets\(\)](#) können die einzelnen Strings später bequem lokalisiert und weiterverarbeitet werden. mit [GetNullOffset\(\)](#) wird die nächste Zeile lokalisiert..

BL muß ein 4 Byte großer Speicher sein, hierin schreibt GetAllSelected() ein LongInt, das Handle des Listviews, in dem Zeilen markiert wurden.

Jede neue Markierung, die ausgewertet wurde, wird von der Listview.dll abgehakt, sodaß nicht mehrmals auf die gleiche Markierung abgefragt werden kann / muß.

Hier ein Beispiel, wie eine typische Profan-WaitInput-Schleife aussehen könnte, um vier Listviews abzufragen (Sourcecode ab Profan 7):

( addr(y&) schreibt das Listview-Handle direkt in die Variable y&, mit Profanversionen niedriger als Version 7 muß eine Bereichsvariable verwendet werden, z.B. x&=GetSelected(itemtexts#,bereich#) ... y&=Long(bereich#,0).)

```

While 1
  WaitInput
  Case %key=2:BREAK                      'WaitInput-Schleife verlassen

  x&=GetNeededMemory(listview&,0)       'Testen, wieviel Speicher für den
Bereich nötig ist,                      '
  Dim marked#,x&                        'der die Daten aufnimmt (wichtig,
immer das Listview testen,              '
                                      'das den meisten Text enthält!). Dann Speicher anfordern.

```

```

z&=GetAllSelected(marked#,addr(y&))      'Wurde in einer Listbox markiert?
If z&                                     'Ja...!

x&=GetColumns(y&)                        'Wieviel Spalten hat Listview y&

Case y&=listview&:Print "Listview 1:"      'Listview-Handle auslesen,
Case y&=listview2&:Print "Listview 2:"     'in dem einer oder mehrere
Case y&=listview3&:Print "Listview 3:"     'Texte markiert
Case y&=listview4&:Print "Listview 4:"     'wurden

offset&=0
WhileLoop z&

    Print "Items in markierter Zeile "+Str$(&loop)+": "

    addoffset&=GetNullOffset(marked#+offset&) 'Nächste Offsetadresse im Bereich
    ermitteln (0)

    y&=GetTabOffsets(marked#+offset&,nurso#) 'Alle Offsets einer Zeile auslesen und
    alle Trennzeichen (9)
                                'gegen Null austauschen

    y&=0
    WhileNot x&=y&                'Schleife zum Ausgeben
        text$=String$(marked#,Long(nurso#,y&*4)+offset&) 'aller Item-Texte in
        Print text$                'einer Zeile.
        y&=y&+1
    EndWhile

    offset&=offset&+addoffset&    'Nächsten Offset vorbereiten
    Print
EndWhile
EndIf
Dispose marked#                  'Speicher freigeben

EndWhile

```

### **GetNeededMemory(H,F)**

Ermittelt, wie viel Speicher eine Funktion benötigen wird.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
F : Long - Flag

Rückgabe-Ergebnis: Long - Anzahl Bytes an benötigtem Speicher.

Manche Funktionen arbeiten mit einem Speicherbereich, dessen Größe dem Programmierer nicht immer bekannt ist.

Mit dieser Funktion kann die Größe des Speichers vorher berechnet werden. Es kann aber nicht davon ausgegangen werden, dass der zurückgegebene Wert nachher tatsächlich die exakte Größe benötigten Speichers widerspiegelt, vielmehr ermittelt GetNeededMemory() immer den Maximalwert, der benötigt werden könnte!!!

Für welche Funktion GetNeededMemory() im Einzelnen gilt, kann mit dem Flag F bestimmt werden.

Bisher werden erst zwei Flags verwendet:

```
0 = GetAllSelected\(\)  
1 = ListviewToCsv\(\)  
2 = ListviewToDbf\(\)
```

Beispiel:

```
bytes&=GetNeededMemory(listview&,0)
```

### **GetTabOffsets(BT,BO)**

Tauscht in einem Speicher alle Tab's gegen Nullbytes aus und speichert die gefundenen Offsets der Tabs als Longint's.

BT : Zeiger auf einen Bereich, der mehrere Texte enthält, die durch Tab's (Chr\$(9)) voneinander getrennt sind.

BO : Zeiger auf einen Bereich, der die Offsetwerte aller gefundener Tab's aufnimmt.

Rückgabe-Ergebnis: Long - Anzahl gefundener Tab's.

Manche Funktionen geben mehrere Texte zurück, die mit Tab's voneinander getrennt sind. Um die einzelnen Textpassagen schnell lokalisieren und weiter bearbeiten zu können, tauscht GetTabOffsets() alle Tabs um in Nullbytes, sodass somit einzelne Strings entstehen. Die Offsets der Speicheradressen, an denen die Strings im Speicher beginnen, werden dann als LongInt's hintereinander nach BO geschrieben.

```
GetTabOffsets(itemtexts#,nurso#)
```

### **GetNullOffset(B)**

Sucht in einem Bereich das nächste Nullbyte.

B : Zeiger auf einen Bereich

Rückgabe-Ergebnis: Long - Anzahl Bytes bis zum nächsten Nullbyte.

```
GetNullOffset(bereich#)
```

### **CreateImageList(F,B)**

Erstellt eine Liste mit Icons (Imagelist). Die Icons in dieser Liste können dann in einem Listview benutzt werden.

F : Long - Flags

B : Zeiger auf einen Bereich, der die Anzahl Icons in der Liste empfängt (Long-Int) oder 0.

Rückgabe-Ergebnis: Long - Handle der erstellten Imageliste.

Der Wert in F gibt an, mit was für Icons die Liste gefüllt wird. Hierbei kann die Anzahl der eingesetzten Icons auf unterschiedlichen Systemen verschieden sein.

Die ersten 49 Icons scheinen aber auf allen Systemen vertreten zu sein. Die Flags sind nicht kombinierbar!

Folgende Flags gibt es bisher:!

#### Flag 0:

Die Liste wird mit den Icons der Shell32.dll gefüllt, eine System-Dll. Bei mir (Windows XP) sind das 54 Icons.

Die Auflösung der Icons ist allerdings ziemlich schlecht, das liegt wohl an Windows Umrechnung von Icon -> Bitmap -> Icon:

Besser Flag 1 benutzen...



#### Flag 1:

Die Liste wird mit den Icons der System ImageList gefüllt. Bei mir sind das 95 Icons. Die ersten 49 Icons sind Systemicons, danach folgen diverse Programmicons:



#### Flag 2:

Die Liste wird mit den "Predefined" System-Icons gefüllt. Bei mir sind es folgende 6 Icons:



#### Flag 3:

Eine leere Liste wird erzeugt.

Die mit Flag 1 erstellte ImageList braucht nicht wieder freigegeben zu werden, das erledigt Windows für uns.

Die mit den anderen Flags erzeugten Listen müssen aber freigegeben werden, das geht mit DestroyImageList():

```
Def DestroyImageList(1)!"comctl32.dll","ImageList_Destroy"
```

Beispiel:

```
imagelist&=CreateImageList(2,addr(anzahlicons&))  
Print "Imagelisthandle = "+Str$(imagelist&)  
Print "Anzahl verfügbarer Icons = "+Str$(anzahlicons&)  
DestroyImageList(imagelist&)
```

### **SetImageList(H,I)**

Ordnet einem Listview eine Imageliste mit Icons zu.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

I : Long - Handle einer mit [CreateImageList\(\)](#) erstellten Icon-Liste

Rückgabe-Ergebnis: Long - Handle der vorher zugeordneten Imageliste.

Bevor Icons in Listviews benutzt werden können, muß hiermit eine Imageliste zugeordnet werden.

Wird für I = 0 genommen, dann wird eine frühere Imageliste wieder vom Listview gelöst.

```
SetImageList(listview&,imagelist&)
```

### **SetIcon(H,Z,N)**

Setzt ein Icon an einer bestimmten Zeile in einem Listview ein.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Z : Long - Zeilennummer im Listview, an der das Icon gesetzt werden soll (nullbasierend)

N : Long - Nummer des Icons in der (dem Listview zugeordneten) ImageList (nullbasierend)

Setzt das Icon Nummer N in das Listview H an Zeile Z.

Ist N = -2, dann wird ein leeres (unsichtbares) Icon eingesetzt.

```
SetIcon(listview&,124,3)
```

### **SetIconsFromMem(H,Z,B,A)**

Setzt mehrere Icons in einem Listview ein. Welche Icons verwendet werden, wird aus einem Speicherbereich ermittelt.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
Z : Long - Zeilennummer im Listview, ab der die Icons gesetzt werden sollen (nullbasierend)  
B : Bereich mit Long-Int's, die die Iconnummern der zugeordneten ImageList beinhalten (nullbasierend)  
A : Long - Anzahl Long-Int's in B

Ist ein Long-Int im Speicher B= -2, dann wird ein leeres (unsichtbares) Icon eingesetzt.

Beispiel:

(Setzt 6 Icons in ein Listview ein, beginnend ab Zeile 0)

```
Long bereich#,0=4  
Long bereich#,4=23  
Long bereich#,8=18  
Long bereich#,12=81  
Long bereich#,16=44  
Long bereich#,20=2  
SetIconsFromMem(listview&,0,bereich#,6)
```

### **SetIconsWith(H,Z,N,A)**

Setzt mehrere gleiche Icons ab einer bestimmten Zeile in einem Listview ein.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
Z : Long - Zeilennummer im Listview, ab der die Icons gesetzt werden sollen (nullbasierend)  
N : Long - Nummer des Icons in der (dem Listview zugeordneten) ImageList (nullbasierend), das verwendet werden soll  
A : Long - Anzahl Icons / Zeilen, die gesetzt werden sollen

Ist N = -2, dann werden leere (unsichtbare) Icons eingesetzt.

Beispiel

(Setzt 100 Icons in ein Listview ein, beginnend ab Zeile 0. Es wird Icon Nummer 14 verwendet)

```
SetIconsWith(listview&,0,14,100)
```

### **SetIndex(I)**

Setzt eine neue Indexzeile für SetItem / SItem() und CsvToListview(), oder eine neue Indexspalte für InsertColumn / IColumn() ein.

I : Long - Indexzeile/spalte, ab wo die nächste erzeugte Zeile/Spalte eingesetzt wird (nullbasierend) oder -1

Wird in einem Listview eine neue Zeile oder Spalte erzeugt, dann wird diese Zeile/Spalte standardmäßig unten, bzw. rechts angehängt.

Manchmal muß man sie aber an anderer Stelle einfügen, das geht, indem der Index verändert wird.

I ist die neue Indexzeile/spalte, von wo an ab jetzt neue Zeilen/Spalten eingefügt werden. Bei jeder neu erzeugten Zeile/Spalte wird I danach automatisch um 1 erhöht. Um den augenblicklichen Wert zu ermitteln, kann [GetIndex\(\)](#) verwendet werden.

Mit `SetIndex(-1)` wird der Index wieder in die Ausgangsstellung gebracht, und neue Zeilen/Spalten werden wieder unten, bzw. rechts angehängt..

`SetIndex()` funktioniert mit `SetItem`, [SItem\(\)](#), [CsvToListview\(\)](#), `InsertColumn` und [IColumn\(\)](#).

Wird es mit `InsertColumn` / `IColumn()` verwendet, dann darf für I nur Null eingesetzt werden, wenn das Listview noch keine Spalten besitzt. Sind schon Spalten vorhanden und für I wird Null verwendet, führt das zu Fehlern, es ist also nicht ohne weiteres möglich, ganz links eine neue Spalte einzufügen (Control-technisch bedingt)!

### **GetIndex()**

Ermittelt den Index, mit dem [SetIndex\(\)](#) gerade arbeitet.

Ohne Parameter.

Ergebnis: Long - Wert von `SetIndex()`.

### **GetSelectedLine(H)**

Findet den Index der ersten markierten Zeile.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Rückgabe-Ergebnis: Long - Index der gefundenen Zeile oder -1.

Beispiel:

```
GetSelectedLine(listview&)
```

### **GetColumnWidth(H,I)**

Ermittelt, wie breit im Listview F die Spalte mit dem Index I ist.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

I : Long - Indexspalte (nullbasierend)

Rückgabe-Ergebnis: Long - Breite (in Pixel) von Spalte I

Beispiel:

```
width&=GetColumnWidth(listview&,8)
```

### **GetLineText(H,Z,BT)**

Kopiert alle Itemtexte einer Zeile (eines Listviews) in einen Speicherbereich, getrennt durch Tab's

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
Z : Long - Indexzeile, aus der alle Itemtexte abgerufen werden sollen (nullbasierend)  
BT: Zeiger auf einen Bereich, der die Texte aller Items in Zeile Z empfängt.

Rückgabe-Ergebnis: Long - Anzahl Spalten im Listview H.

BT muß ein Speicherbereich sein, der 16384 Bytes groß sein sollte (max. bis zu 64 Spalten a 256 Bytes Text = 16384).

Hierin schreibt GetLineText() alle Item-Texte der Zeile mit dem Index Z, die einzelnen Texte sind durch Tabs (Chr\$(9)) getrennt, das allerletzte Byte ist ein Nullbyte.

Mit [GetTabOffsets\(\)](#) können die einzelnen Strings später bequem lokalisiert und weiterverarbeitet werden.

Beispiel:

```
columns&=GetLineText(listview&,8,itemtexts#)
```

### **CryptMem (B,A,S,L)**

Verschlüsselt Daten in einem Speicherbereich mit einem beliebig langen Passwort. Bei zweimaligem gleichem Aufruf sind die originalen Daten wieder hergestellt.

B : Long - Bereichsvariable, in der die Daten gespeichert sind  
A : Long - Anzahl zu verschlüsselnder Bytes  
S : Long - Zeiger auf einen String (oder Bereich), in dem das Passwort gespeichert ist  
L : Long - Länge des Passworts

Eine superschnelle und relativ sichere Datenverschlüsselung auf XOR-Passwort-Basis.

Wichtig z.B. für CSV-Dateien, die ja ansonsten sehr einfach einzusehen und zu manipulieren sind.

Beispiel um eine verschlüsselte CSV-Datei in ein Listview zu laden und später wieder zurück zu speichern:

```
name$="Datei.csv"
x&=@FileSize(name$)

ReadFileQuick(addr(name$),bereich#,0,x&)           'Datei in Speicherbereich
einladen                                           'Datei in Speicherbereich

passwort$="Beliebiges Passwort"                   'Passwort
CryptMem(bereich#,x&,addr(passwort$),len(passwort$)) 'Daten in dem Bereich
entschlüsseln                                     'Daten in dem Bereich

CsvToListview(listview&,bereich#,x&,8)           'CSV-Datei in Listview einlesen

[...]

x&=ListviewToCsv(listview&,bereich#,0,0)          'Listview-Daten im CSV-Format
auslesen                                           'Listview-Daten im CSV-Format

passwort$="Beliebiges Passwort"                   'Passwort
CryptMem(bereich#,x&,addr(passwort$),len(passwort$)) 'Daten in dem Bereich wieder
verschlüsseln                                     'Daten in dem Bereich wieder

name$="Datei.csv"
WriteFileQuick(addr(text$),bereich#,0,x&)          'Daten wieder als Datei
speichern                                           'Daten wieder als Datei
```



## **CsvToHeader(H,B,O)**

Automatisches Erstellen von Spalten aus dem Header einer CSV-Datei.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen Bereich, in den mit [ReadFileQuick\(\)](#) eine CSV-Datei geladen wurde.  
O : Zeiger auf einen Bereich, der den neuen Offset innerhalb von B empfängt (Long-Int) oder 0.

Rückgabe-Ergebnis: Long - Anzahl generierter Spalten im Listview H.

Schneller und bequemer kann man die Spalten eines Listviews nicht mehr generieren.

Die erste (sinnvolle) Zeile einer [CSV-Datei](#), die ja Trennzeichen enthält, wird dazu verwendet, Spalten und Spaltenbuttons in einem Listview zu generieren. Nach jedem Trennzeichen wird die nächste Spalte erzeugt und mit einer Überschrift gefüllt wird.

O sollte ein Zeiger auf eine Variable oder ein 4 Byte großer Speicherbereich sein. Diese Variable (Speicher) empfängt nachher den neuen Offset innerhalb von B, an dem die nächsten CSV-Daten beginnen, ohne die Header-Zeile. Sinnvoll, wenn anschließend [CsvToListview\(\)](#) erfolgt, es braucht nur noch der Offset zum Startpunkt der Daten addiert werden, und von der Anzahl Daten in Bytes abgezogen werden. Ist O = Null, dann wird diese Variable nicht übermittelt.

Beispiel, um eine CSV-Datei einzulesen, Spalten in einem Listview damit zu generieren und anschließend die Items innerhalb des Listviews:

```
text$="Datei.csv"
bytes&=@FileSize(text$)

Dim bereich#,bytes&

ReadFileQuick(addr(text$),bereich#,0,bytes&) ;Datei laden
spalten&=CsvToHeader(listview&,bereich#,addr(offset&)) ;Spalten und
Spaltenbuttons generieren
CsvToListview(listview&,(bereich#+offset&),(bytes&-offset&),spalten&) ;Items und
Itemtexte generieren

Dispose bereich#
```

Es sei noch zu erwähnen, das die Breite der Spalten sich an der Breite der Spaltenbutton-Texte orientiert. Die Breite kann nachträglich mit

[SetColumnWidth\(listview&,Spaltenindex&,Pixelbreite&\)](#) geändert werden.

Als Textausrichtung wird automatisch linksbündig (Text) gewählt, enthält der Spaltenbutton-Text nur bestimmte Ziffern, dann wird als Textausrichtung rechtsbündig gewählt (Zahl). Diese Ziffern sind:

0 1 2 3 4 5 6 7 8 9 0 . , -

Sollte die Textausrichtung nachträglich verändert werden müssen, dann kann hierzu die Funktion [SetColumnAlignment\(\)](#) verwendet werden.

## **DbfToCsv(B,A,S,Z,F)**

Wandelt eine dBaseIII (.dbf) Datei um in eine [CSV-Datei](#) (im Speicher).

B : Zeiger auf einen Bereich, in den mit [ReadFileQuick\(\)](#) eine DBF-Datei geladen wurde.

A : Long - Anzahl Bytes in B.  
S : Zeiger auf einen Bereich, der die Anzahl Spalten in der Dbf-Datei in B empfängt (Long-Int) oder 0.  
Z : Zeiger auf einen Bereich, der die Anzahl Zeilen in der Dbf-Datei in B empfängt (Long-Int) oder 0.  
F : Long - Flag.

Rückgabe-Ergebnis: Long - Anzahl generierter Bytes in der neuen Csv-Datei (in B), bei Fehler 0.

Weil viele User gerne mit dBase-Dateien arbeiten, wurde diese Funktion implementiert.

A ist die Größe der Datei in B.

S muß eine Variable sein oder ein vier Byte großer Speicher, in die nachher die Anzahl Spalten eingetragen ist. steht hier Null, dann wird der Parameter S ignoriert.

Z ist ebenfalls eine Variable oder ein vier Byte großer Speicher, nur steht hier nachher die Anzahl Zeilen.

F ist ein Flag. Ist Bit Null gelöscht, dann werden auch die Felder (Headerinformationen) der Dbf-Datei mit ausgelesen. Die erste Zeile in der Csv-Datei beinhaltet dann die Feldnamen. Ist Bit Null in F gesetzt, dann werden die Felder ignoriert und nicht mit in die Csv-Datei übernommen. Feldnamen sind vergleichbar mit den Texten der Spaltenbuttons.

Ist Bit Eins in F gesetzt, dann werden Datumsfelder vom Type 20020726 automatisch eingesetzt als 2002.07.26 (=26.07.2002).

Ist Bit Zwei gesetzt, dann werden alle Texte vom OEM-Format in das ANSI-Format konvertiert.

Hier nochmal die Liste aller möglichen Flags ( kombinierbar durch OR Verknüpfung oder Addition).

0 = Felder nicht mit in die Csv-Datei auslesen,  
keine Datumsfelder-Konvertierung,  
OEM-nach-ANSI Konvertierung,

1 = Felder mit in die Csv-Datei auslesen

2 = Datumsfelder konvertieren, z.B. 20020726 wird zu 2002.07.26

4 = keine OEM-nach-ANSI-Konvertierung (betrifft nur die Buchstaben Ä, Ö, Ü, ä, ö, ü, ß)

Die neu erstellte Csv-Datei steht nachher in B, die alten Daten in B werden überschrieben. Weil eine Dbf-Datei quasi immer größer ist, als eine Csv-Datei, muß bei der Speicherbereichs-Größe normalerweise nichts beachtet werden.

Ist das Ergebnis der Funktion 0, trat ein Fehler auf, oder die Datenbank war noch leer.

```
text$="Datei.dbf"
bytes=&@FileSize(text$) 'Größe der Datei ermitteln
(bytes&)
If bytes&>0
  Dim bereich#,bytes&
  ReadFileQuick(addr(text$),bereich#,0,bytes&) 'Dbf-Datei laden
  bytes&=DbfToCsv(bereich#,bytes&,addr(spalten&),0,0) 'Dbf konvertieren nach Csv /
bytes& neu ermitteln
  If bytes&
    CsvToListview(listview&,bereich#,bytes&,spalten&) 'Csv Datei als Items in Listview
einlesen
  EndIf
  Dispose bereich#
EndIf
```

### **SelectLine(H,I,F)**

Selektiert oder Deselektiert eine oder alle Zeilen eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Z : Long - Indexzeile, die selektiert werden soll (nullbasierend).  
F : Long - Flags.

Mit dieser Funktion kann entweder eine oder alle Zeilen selektiert / deselektiert werden. Auf Wunsch scrollt das Listview in die betroffene Zeile.

Hier eine Liste aller möglichen Flags ( kombinierbar durch OR Verknüpfung oder Addition).

0 = nur eine Zeile (Z) selektieren / nicht in die Zeile scrollen  
1 = Zeile Z in den sichtbaren Bereich scrollen  
2 = nicht selektieren, sondern deselektieren (entmarkieren)  
4 = Z ignorieren und alle Zeilen bearbeiten  
8 = [GetAllSelected\(\)](#) zwingen, auch mit SelectLine() markierte Zeilen aufzulisten.

In seltenen Fällen kann es beim Gebrauch von SelectLine() zu ungewollten Scroll-Phänomen kommen. Lesen sie in diesem Fall nach unter [ForbidScrollMessages\(\)](#).

Beispiele:

```
SelectLine(listview&,8,0)           ;Selektiert Zeile 8
SelectLine(listview&,8,2)           ;Deselektiert Zeile 8
SelectLine(listview&,100,1)         ;Selektiert Zeile 100 und scrollt zu Zeile 100
SelectLine(listview&,100,3)         ;Deselektiert Zeile 100 und scrollt zu Zeile 100
SelectLine(listview&,0,4)           ;Selektiert alle Zeilen
SelectLine(listview&,68,5)          ;Selektiert alle Zeilen und scrollt zu Zeile 68
SelectLine(listview&,0,12)          ;Selektiert alle Zeilen und aktualisiert
GetAllSelected()
SelectLine(listview&,120,7)         ;Deselektiert alle Zeilen und scrollt zu Zeile 120
SelectLine(listview&,0,6)           ;Deselektiert alle Zeilen
```

### **SearchText(H,SZ,EZ,SS,S,F,V)**

Sucht einen Text innerhalb des Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
SZ: Long - Index der Startzeile, ab der die Suche beginnt (nullbasierend).  
EZ: Long - Index der Endzeile, bei der die Suche endet (nullbasierend).  
SS: Long - Index der Startspalte, ab der die Suche beginnt (nullbasierend).  
S : Zeiger auf den Suchstring (Adresse eines Strings oder Bereichsvariable)  
F : Long - Flags  
V : Zeiger auf eine Variable oder 4 Byte großer Bereich, der den Index der ersten gefundenen Spalte, die den Suchtext enthält, empfängt (Long-Int) oder 0.

Rückgabe-Ergebnis: Long - Index der ersten gefundenen Zeile, die den Suchtext enthält, oder -1.

Eine komfortable Suchfunktion, um ein Wort oder Teilwort in einem Listview zu finden.

SZ ist die Zeile, ab der die Suche startet, bei Zeile EZ endet die Suche schließlich.

In SS kann die Spalte angegeben werden, ab der die Suche beginnt. SS ist außerdem zusammen mit F = 2 von Bedeutung.

S muß ein Zeiger auf einen String sein, oder eine Bereichsvariable, in die der Suchstring zuvor geschrieben wurde.

Der Suchstring S muß mit einem Nullbyte abgeschlossen sein (ist bei allen Strings ab Profan 7 der Fall).

V muß ein 4 Byte großer Speicher sein, hierin schreibt SearchText() ein LongInt: Den Index der ersten gefundenen Spalte, die den Suchtext enthält.

Ab Profan 7 kann so direkt in eine Variable geschrieben werden. Wenn für V beispielsweise Addr(spalte&) verwendet wird, so steht nachher in der Variablen spalte& die betreffende Spalte, in der der Suchtext als erstes gefunden wurde, oder -1, wenn der Suchtext nicht gefunden wurde.

Verwenden sie für V eine Null, dann wird dieser Parameter ignoriert.

In F können einige Flags angegeben werden. Hier eine Liste aller möglichen Flags ( kombinierbar durch OR Verknüpfung oder Addition).

0 = Groß-Klein-Schreibung nicht beachten (z.B. Frank=frANK) / jede Spalte durchsuchen / Suchtext überall erlaubt.  
 1 = Groß-Klein-Schreibung beachten (z.B. Frank<>frANK)  
 2 = Der Suchtext wird ausschließlich nur in Spalte SS gesucht (Textsuche für nur eine Spalte)  
 4 = Der Suchtext muß am Anfangs eines Worts stehen, damit er gefunden wird.  
 8 = Der Suchtext muß dem kompletten Itemtext entsprechen, und darf nicht nur ein Teilstück sein.

Auch Zahlen werden innerhalb des Listviews als Text gesucht / gefunden. Das ListView verwaltet alle Texte / Zahlen usw. ausschließlich als Texte.

Um freie Einträge (leere Itemtexte) zu finden kann SearchText() nicht verwendet werden. Dafür ist [SearchBlankItem\(\)](#) zuständig.

Beispiel 1 (Findet den ersten Suchstring):

```
such$="Text"
y&=SearchText(listview&,0,GetLines(listview&),0,addr(such$),0,addr(x&))
Case (y&<>-1):Print "Gefunden, Zeile "+Str$(y&)+" / Spalte "+Str$(x&)
```

Beispiel 2 (Findet alle Suchstrings):

```
such$="Text"
x&=0
y&=0
While 1
  y&=SearchText(listview&,y&,GetLines(listview&),x&,addr(text$),0,addr(x&))
  Case y&=-1:BREAK
  Print "Gefunden, Zeile "+Str$(y&)+" / Spalte "+Str$(x&)
  x&=x&+1
Wend
```

Beispiel 3 (Findet alle Suchstrings in Spalte 4):

```
such$="Text"
y&=0
While 1
  y&=SearchText(listview&,y&,GetLines(listview&),4,addr(text$),2,addr(x&))
  Case y&=-1:BREAK
  Print "Gefunden, Zeile "+Str$(y&)+" / Spalte "+Str$(x&)
  y&=y&+1
Wend
```

## ListviewToDbf(H,B,A,F)

Auslesen eines Listviews in eine dBaseIII-Datei (.dbf).

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten ListView Controls  
 B : Zeiger auf einen Bereich, in den alle ListView-Texte als DBF-Datei gespeichert werden  
 A : Long - Anzahl verfügbarer Bytes in B (ermittelt durch [GetNeededMemory\(\)](#))  
 F : Long - Flag (0 = Ansifont / 1 = Oemfont).

Ergebnis: Long - Anzahl Bytes, die in den Bereich B geschrieben wurden.

Eine Funktion, um den Inhalt eines Listviews abzuspeichern. ListViewToDbf() schreibt den Inhalt des Listviews H als dBaseIII-Datei in den Speicher B. Der Speicher muß natürlich groß genug dimensioniert sein, wie groß er sein muß, kann vorher mit [GetNeededMemory\(\)](#) ermittelt werden....

Mit [WriteFileQuick\(\)](#) lässt er sich dann auf einem Datenträger abspeichern.

Mit F lässt sich bestimmen, ob Sonderzeichen (deutsche Umlaute und ß ) als Ansi- oder Oem-Font gespeichert werden.

Alle Itemtexte eines Listview stehen dort als Texte (Strings). Darum speichert ListViewToDbf() in der .dbf-Datei auch alle Daten als Textfeld (Buchstabe C) ab.

Um numerische oder sonstige Werte korrekt in die .dbf-Datei zu schreiben, müssen sie den Datentyp selber von Hand ändern. Hier ein Beispiel mit 5 Spalten, wobei die erste, vierte und fünfte numerisch sind, die zweite und dritte enthalten Texte.

```
bytes&=GetNeededMemory(listview&,2)
Dim bereich#,bytes&

bytes&=ListViewToDbf(listview&,bereich#,bytes&,0)
Char bereich#,43="N"
Char bereich#,(43+32)="C"      'eigentlich unnötig, enthält schon C
Char bereich#,(43+64)="C"      'eigentlich unnötig, enthält schon C
Char bereich#,(43+96)="N"
Char bereich#,(43+128)="N"

text$="Datei.dbf"
WriteFileQuick(addr(text$),bereich#,0,bytes&)

Dispose bereich#
```

### **CopyColumnTo(H,Z,I,E)**

Kopiert eine komplette Spalte von einem Listview in ein anderes Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls (Quell-Listview)  
Z : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls (Ziel-Listview)  
I : Long - Index der Spalte von H, die kopiert werden soll (nullbasierend).  
F : Long - Index der Spalte in Z, an die die Spalte kopiert werden soll (nullbasierend).

Kopiert die Spalte I aus dem Listview H an Position F in Listview Z. Eventuell rechts daneben liegenden Spalten werden nach rechts hin verschoben.

Die komplette Spalte mit Itemtexten (aber ohne evt. vorhandene Icons) wird kopiert. Besitzt Listview Z nicht so viele Zeilen wie H, dann werden so viele neue Zeilen erzeugt, wie nötig sind um alle Texte zu erfassen.

Sind in Z noch keine Spalten vorhanden, dann sollte F = Null (0) sein. Gibt es in Z aber schon Spalten, dann sollte für F nicht Null verwendet werden, das führt zu Fehlern. Es ist also nicht ohne weiteres möglich, eine neue Spalte ganz links einzufügen (Control-technisch bedingt)!

Es könnte in diesem Fall aber zu Beginn eine leere "Dummyspalte" erzeugt werden, die nach den Kopiervorgängen wieder gelöscht wird:

```
IColumn(listview&,addr(text$),1,0)
...
DeleteColumn(listview&,0)
```

Beispiel:

```
CopyColumnTo(listview&,listview2&,8,3)
```

### **ExchangeSeparator(B,A,AT,NT,F)**

Tauscht einzelne Bytes gegen andere aus. Dazu gedacht, um CSV-Dateien mit beliebigen Trennzeichen verarbeiten zu können.

B : Long - Bereich#, in dem die Daten stehen, die ausgetauscht werden sollen.  
A : Long - Anzahl Bytes, die in A überprüft werden sollen  
AT: Long - ASCII Code der Trennzeichen, die überschrieben werden sollen.  
NT: Long - ASCII Code der Trennzeichen, mit denen AT überschrieben wird.  
F : Long - Flag

Die Listview.dll kann nur CSV-Dateien verarbeiten, die als Trennzeichen ein Komma oder Semikolon verwenden. Mit ExchangeSeparator() ist es jetzt möglich, Csv-Dateien (B) mit beliebigen Trennzeichen zu verwenden, indem vor Benutzung der Datei einfach die Trennzeichen ausgetauscht werden.  
Alle Bytes/Texte die innerhalb von Anführungszeichen stehen, sind davon nicht betroffen, wenn F=0 ist.

---

**Ab Version 1.7 ist ein Flag dazu gekommen. Ist F=1, dann sind auch diejenigen Bytes/Texte vom Austausch betroffen, die innerhalb von Anführungszeichen stehen!  
Bitte ältere Quelltexte um den Zusatzparameter erweitern!**

---

Beispiel, tauscht alle Rauten (#) um in Semikolons (;):

```
ExchangeSeparator(bereich#,bytes&,@Ord("#"),@Ord(";"),0)
```

### **CopyLineTo(H,Z,I,F)**

Kopiert eine komplette Zeile von einem Listview in ein anderes Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls (Quell-Listview)  
Z : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls (Ziel-Listview)  
I : Long - Index der Zeile von H, die kopiert werden soll (nullbasierend).  
F : Long - Index der Zeile in Z, an die die Zeile kopiert werden soll (nullbasierend).

Kopiert die Zeile I aus dem Listview H an Position F in Listview Z. Eventuell darunter liegenden Zeilen werden nach unten hin verschoben.

Die komplette Zeile mit Itemtexten (aber ohne evt. vorhandene Icons) wird kopiert. Besitzt Listview Z nicht so viele Spalten wie H, dann werden so viele neue Spalten (incl. Spaltenbutton-Texte) erzeugt, wie nötig sind um alle Texte zu erfassen.

In Gegensatz zu [CopyColumnTo\(\)](#) können auch Zeilen über der ersten Zeile eingefügt werden, da gibt es hier keine Einschränkungen.

```
CopyLineTo(listview&,listview2&,8,3)
```

### **GetChecked(BL,BI,BS)**

Ermittelt, ob und in welcher Zeile welchen Listviews zuletzt eine Checkbox angeklickt wurde.

BL : Zeiger auf einen Bereich, das Handle des angeklickten Listviews empfängt.  
BI : Zeiger auf einen Bereich, der den Index (nullbasierend) der angeklickten Zeile im Listview BL empfängt  
BS : Zeiger auf einen Bereich, der den Status der Checkbox in Zeile BI des Listviews BL empfängt

Rückgabe-Ergebnis: Long - Wenn eine Checkbox angeklickt wurde 1 oder, wenn seit der letzten Abfrage keine Checkbox angeklickt wurde, 0.

Wieder eine Funktion, die dem User einiges an Arbeit abnimmt.

BL muß ein 4 Byte großer Speicher sein, hierin schreibt GetChecked() ein LongInt, das Handle des Listviews, das angeklickt wurde.

BI muß ein 4 Byte großer Speicher sein, wo hinein die Funktion ein LongInt schreibt, den Index der Zeile, in der die Checkbox das angeklickt wurde.

BS muß ein 4 Byte großer Speicher sein, hierin schreibt die Funktion ein LongInt, den Status der angeklickten Checkbox (nach dem Anklicken).

Status:

0 = nicht markiert

1 = Checkbox ist mit einem Häkchen markiert

Jeder Checkbox-Klick, der mit GetChecked() ausgewertet wurde, wird von der ListView.dll abgehakt, sodaß nicht mehrmals auf den gleichen Klick abgefragt werden kann / muß.

Um überhaupt Checkboxes in Listviews verwenden zu können, muß Style 4 der LVS\_EX\_STYLES gesetzt werden.

Mehr Infos hierzu unter [CreateListView\(\)](#).

Hier ein Beispiel, wie eine typische Profan-WaitInput-Schleife aussehen könnte, um die Checkboxes von Listviews abzufragen (Sourcecode ab Profan 7):

( z.B. addr(handle&) schreibt das ListView-Handle direkt in die Variable handle&, mit Profanversionen niedriger als Version 7 muß eine Bereichsvariable verwendet werden, z.B. Dim

bereich#,12...x&=GetChecked(bereich#,bereich#+4,bereich#+8)

...handle&=Long(bereich#,0)...indexline&=Long(bereich#,4)...status&=Long(bereich#,8)...Dispose bereich#...

```
x&=GetChecked(addr(handle&),addr(indexline&),addr(status&))
If x&
  Print handle&      'Handle eines Listviews
  Print indexline&   'Zeilennummer
  Print status&      'Status, 0 oder 1
EndIf
```

### **SetCheckboxState(H,I,S)**

Markiert oder entmarkiert eine (oder alle) Checkbox(en) in einem ListView-Control.

H : Long - Handle eines mit [CreateListView\(\)](#) erstellten ListView Controls

I : Long - Index der Zeile von H, in der die Checkbox markiert / entmarkiert werden soll (nullbasierend).

S : Long - Status der Markierung (0 = entmarkieren / 1 = markieren).

Markiert / entmarkiert Checkboxes innerhalb eines Listviews. Natürlich nur, wenn Checkboxes vorhanden sind...

Um Checkboxes in Listviews verwenden zu können, muß Style 4 der LVS\_EX\_STYLES gesetzt werden. Mehr Infos hierzu unter [CreateListView\(\)](#).

Ist S = 1, dann wird die jeweilige Checkbox mit einem Häkchen versehen, wenn S = 0 ist, wird das Häkchen entfernt.

Wenn I = -1 ist, werden alle Checkboxes markiert / entmarkiert.

```
SetCheckboxState(listview&,8,1)
```

### **GetCheckboxState(H,I)**

Ermittelt, ob eine Checkbox in einem ListView-Control markiert oder entmarkiert ist.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
I : Long - Index der Zeile von H, in der der Status einer Checkbox ermittelt werden soll (nullbasierend).

Ergebnis: Long - Status der Markierung, 0 = entmarkiert / 1 = markiert

Ermittelt den Status einer Checkbox innerhalb eines Listviews. Natürlich nur, wenn Checkboxes vorhanden sind... Um Checkboxes in Listviews verwenden zu können, muß Style 4 der LVS\_EX\_STYLES gesetzt werden. Mehr Infos hierzu unter [CreateListview\(\)](#).

Ist das Ergebnis der Funktion = 1, dann ist die jeweilige Checkbox mit einem Häkchen markiert, wenn das Ergebnis 0 ist, existiert kein Häkchen in der Checkbox.

```
x&=GetCheckboxState(listview&,8)
If x&=1
    Print "Checkbox ist markiert."
Else
    Print "Checkbox ist nicht markiert."
Endif
```

### **EnableEdits(H,F)**

Erlaubt in einem Listview editierbare Itemtexte (oder auch nicht).

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
F : Long - Flag

Normalerweise arbeiten Listview-Controls ohne editierbare Itemtexte.

Mit EnableEdits() kann dieses Feature aber eingeschaltet (oder wieder abgeschaltet) werden, und zwar für jedes einzelne Listview.

Ob ein Listview editierbar werden soll, bestimmt das Flag F.

```
0  = H soll kein editierbares Listview sein
1  = H soll editierbar sein
2  = Sobald ein Editfeld erscheint, wird der Text darin selektiert (ohne Flag 2
bleibt
    der Text normal)
4  = Editieren startet mit linkem Maus-Doppelklick
8  = Editieren startet mit rechtem Mausklick und linkem Maus-Doppelklick
16 = Springen mit Cursortaste links/rechts und komplettes Text-Entfernen mit "Entf"
verbieten
```

Bisher waren die Flags 2-16 global. Ab Listview.dll-Version 1.7 sind sie jetzt aber auch lokal und können für jedes Listview unterschiedlich eingestellt werden.

Ein Editfeld wird in einem Listview erzeugt, sobald der Anwender die rechte Maustaste (default) über einem Itemtext drückt.

Es erscheint das Editfeld mit dem Itemtext, der jetzt editiert werden kann.

Benutzt der Anwender die rechte Maustaste zusammen mit der Taste STRG, dann erscheint das Editfeld mit dem Inhalt der Zwischenablage (erste Textzeile, die im Clipboard gespeichert ist).

Ist Flag 4 gesetzt, dann ist anstatt des rechten Mausklicks der linke Maus-Doppelklick aktiv.

Ist Flag 8 gesetzt, dann sind beide Mausklicks gestattet. Flag 4 und 8 dürfen nicht zusammen benutzt werden!

Ist das Editfeld nun erschienen, dann kann solange editiert werden, bis eine Maustaste oder eine der Tasten Return, Tab, Bild hoch, Bild runter, Pos 1, Ende oder die Cursortasten gedrückt wurde. Danach verschwindet das Editfeld und setzt den editierten Text als Itemtext an passender Stelle in das Listview.

Wurde die Tabtaste gedrückt, dann baut sich anschließend ein neues Editfeld auf, sodas der nächste Itemtext



editiert werden kann.

Wenn die Tabtaste zusammen mit SHIFT gedrückt wurde, kann der vorhergehende Itemtext editiert werden. Die Tab-Taste kann jedoch auch unter mehreren Controls hin und her springen. Darum empfehlen sich immer die Cursortasten, um zu springen.

Wird eine Cursortaste gedrückt, dann wird das Editfeld in diese Richtung weitergesetzt. Bei den Cursortasten links/rechts kann man das per Flag 16 aber verhindern. Ist Flag 16 nicht gesetzt, kann mittels der Entf-Taste der komplette Text im Editfeld gelöscht werden.

Wenn der User die Spalten per Drag&Drop verschoben hat, kann sich die Reihenfolge innerhalb des Spaltenindexes ändern!

Ein Trackmenü kann während des Editierens nicht benutzt werden, dafür aber die üblichen Windows Tastenkombinationen:

```
STRG+C = Kopieren  
STRG+X = Ausschneiden  
STRG+V = Einfügen  
STRG+Z = Rückgängig  
ENTF = Löschen  
STRG+SHIFT+CURSORTASTE = Textblock markieren.
```

**Anmerkung:** Windows Listview-Controls unterstützen normalerweise nicht das Editieren in beliebigen Spalten, lediglich in der ersten Spalte kann editiert werden. Hierbei reagiert das System-Editcontrol genauso, wie die Editcontrols, welche die Listview.dll erzeugt. D.h. auch dort unterbricht ein Mausklick die Eingabe. Ich selber hätte gern zusätzlich zu den Editfeldern noch ein Trackmenu gestattet, aber das scheint nicht möglich zu sein, weil das Editcontrol nur ein Childwindows des Listview-Controls ist....

Editierbare Listview mit Profan sollten nur in einem Fenster mit Dialogstyle (Fensterstyle 512) aufgerufen werden.

Ab Listview.dll Version 1.4 kann mit [SelectColumnEdits\(\)](#) entschieden werden, das nur bestimmte Spalten editierbar sind.

Mit Version 2.0 kamen manuell erzeugte Editfelder dazu, um komfortable Eingabemasken programmieren zu können. Diese werden mit [EditManual\(\)](#) erzeugt. EnableEdits() sollte deaktiviert sein/werden, wenn die manuellen Felder benutzt werden sollen.

### **SortManual(H,S,F)**

Sortiert ein Listview-Control "von Hand".

```
H : Long - Handle eines mit CreateListview\(\) erstellten Listview Controls  
I : Long - Index der Spalte von H, die sortiert werden soll (nullbasierend).  
F : Long - Flag
```

Mit [ASortListview\(\)](#) kann eine durch den User durchgeführte Sortierung eines Listviews aktiviert werden. Manchmal kann es aber auch sinnvoll sein, das der Programmierer selbst eine Sortierung vornehmen kann. Genau hierzu dient SortManual().

Voraussetzung ist aber, das die Sortierung für die Spalte I aktiviert wurde!

I ist die Nummer der Spalte, die sortiert werden soll, F ist ein Flag, was die Sortierrichtung bestimmt:

```
0 = Listview wird aufsteigend sortiert  
1 = Listview wird absteigend sortiert
```

Ob eine Sortierung nach Buchstaben oder nach Zahlen erfolgt, wurde ja schon vorher mit ASortListview() bestimmt.

```
SortManual(listview&,8,1)
```

### **FilelistToCsv(N,B,S,Z,F,I)**

Lädt ein Dateiverzeichnis ein und speichert die Daten als Csv-Datei.

N : Zeiger auf einen String mit dem Namen (z.B. "C:\Programme") des Dateiverzeichnisses.

B : Zeiger auf einen Bereich, in dem alle Ordernamen/Dateinamen als CSV-Datei gespeichert werden.

S : Zeiger auf eine Variable oder 4 Byte großer Bereich, der die Anzahl generierter Spalte empfängt (Long-Int) oder 0.

Z : Zeiger auf eine Variable oder 4 Byte großer Bereich, der die Anzahl generierter Zeilen empfängt (Long-Int) oder 0.

F : Long - Flags

I : Zeiger auf einen Speicherbereich, der eine Icon-Index-Tabelle empfängt oder 0.

L : Zeiger auf eine Imageliste oder 0.

Ergebnis: Long - Anzahl generierter Bytes in der Csv-Datei, bei Fehler = -1.

Eine mächtige Funktion, um ganze Dateiverzeichnisse (Name und Icons, Größe, Typ, Letzte Änderung und/oder Attribute) indirekt in ein Listview-Control zu laden, genauso gut kann die Csv-Datei aber auch nur gespeichert werden:

Name	Größe	Typ	Geändert am	Attribute
Icon		Dateiordner	2002\06\08 11:20	
Include		Dateiordner	2002\06\08 11:20	
Lib		Dateiordner	2002\06\08 11:20	
Templates		Dateiordner	2002\06\08 11:20	
_DEISREG.ISR	148	ISR-Datei	2002\06\08 11:20	
_ISREG32.DLL	36352	DLL-Datei	1997\03\07 18:00	
Copying.txt	18581	TXT-Datei	1999\02\02 11:35	
Debugger.txt	905	TXT-Datei	2000\06\11 15:45	
DeIsL1.isu	33987	ISU-Datei	2002\06\08 11:20	
DevCpp.exe	2232320	EXE-Datei	2000\09\21 23:22	
devcpp.ini	3142	INI-Datei	2003\03\13 22:02	A
DOSCOM.EXE	41984	EXE-Datei	2000\01\21 18:22	
Minaw and Cwawin.txt	2054	TXT-Datei	2000\04\27 19:38	

N ist der Zeiger auf einen String. Hier muss ein gültiger Dateipfad angegeben werden, z.B.:

A:

C:\Programme

D:\Spiele\Asteroids

B muß ein Zeiger auf ein Speicherbereich sein. Die Größe kann vorher nicht berechnet werden, das würde den Programmfluß ansonsten behindern. Aber als Faustregel kann man sagen, das pro 1000 eingelesener Ordner/Dateien ca. 100.000 Bytes benötigt werden (wenn alle Flags aktiviert sind). Wer auf Nummer sicher gehen will, übergibt 1 MB Speicher, das sollte mindestens für über 10000 Ordner/Dateien reichen. Solche Riesen-Directories sind aber extrem selten...

Mit S und Z kann ein Zeiger auf eine Long-Int Variable (4 Byte großer Speicher) übergeben werden. In der Variable steht nachher die Anzahl Spalten bzw. Zeilen, die in der Csv\_Datei generiert wurden. Steht in S und/oder Z eine 0, dann wird dieser Parameter ignoriert.

F ist ein Long-Int, das die Flags repräsentiert. Diese Flags gibt es bisher (kombinierbar durch OR Verknüpfung oder Addition):

\$0 = kein Flag gewählt

\$1 = Spalte für Dateigröße (Größe) in die Csv-Datei hinzufügen

\$2 = Spalte für Dateityp (Typ) in die Csv-Datei hinzufügen

\$4 = Spalte für letztes Dateidatum (Geändert am) in die Csv-Datei hinzufügen

\$8 = Spalte für Dateiattribute (Attribute) in die Csv-Datei hinzufügen

\$10 = Index aller verwendeten Icons in I eintragen (nur sinnvoll, wenn Icons benötigt werden)

\$100 = keine Headerzeile (Name | Größe | Typ | ...) in die Csv-Datei einfügen

\$200 = nicht mehr belegt

\$400 = Versteckte Dateien (Hidden Flag) sollen nicht angezeigt werden

I ist ein Zeiger auf einen Speicher, der benötigt wird, wenn Flag \$10 gesetzt ist. Dieser Speicher wird später für [SetIconsFromMem\(\)](#) benötigt, um die ermittelten Icons zuzuweisen. Pro Zeile werden 4 Bytes Speicher benötigt. Wenn sie keine Icons verwenden wollen, setzen sie hier einfach 0 ein.

Ebenfalls muss in L der Handle einer Imageliste stehen, wenn Flag \$10 gesetzt ist. FilelistToCsv leert diese Liste bei jedem Aufruf und bestückt sie neu mit allen Icons, die für die Anzeige der Filelist-Icons später benötigt werden. Die Imageliste L muss in jedem Fall mit dem Listview verknüpft sein, in welches die Anzeige der Liste erfolgen soll. Sollen gleichzeitig in mehrere Listviews Filelisten inclusive Icons eingelesen werden (z.B. für die Programmierung eines Dateimanagers a la Norton Commander), so muss jedem Listview eine eigene Imageliste zugewiesen sein!

Ab Version 1.4 gibt es auch Dateifilter. Mehr hierzu unter [SetFilelistFilter\(\)](#) und [SetFilelistNoFilter\(\)](#).

Die Dateifilter müssen natürlich bestimmt werden, bevor FilelistToCsv() aufgerufen wird...

Einiges gilt es bei der späteren Auswertung der generierten Csv-Datei zu beachten.

Vor den Ordner-/Dateinamen wird ein Byte eingefügt, bei Ordnern das Byte 32 (Leerzeichen), bei Dateinamen das Byte 160 (SHIFT+Leerzeichen).

Das ist deswegen so, damit später im Listview beim Sortieren Ordner und Dateien immer getrennt werden können. Ausserdem kann der Programmierer so allein vom Namen ableiten, ob es sich um einen Ordner oder eine Datei handelt.

Um mit dem Namen also zu arbeiten, muß die Textlänge um 1 gekürzt werden und der Zeiger auf den Text um ein Byte erhöht werden.

Bei der Dateigröße wird bei Ordnern ebenfalls ein Byte eingetragen, 160 (SHIFT+Leerzeichen). Ebenfalls aufgrund der Listview-Sortierung. Bei Dateien wird aber nur die Größe eingetragen, ohne Extra-Byte.

Für "Geändert am" wird folgendes Format verwendet: Jahr-Monat-Tag Stunde:Minute, also z.B. 2003-03-28 23:12.

Das hat ebenfalls den Vorteil, das eine Sortierung nach Dateidatum vorgenommen werden kann!

Bei Attribute werden folgende Attribute angezeigt:

A = Archiv-Datei (archive)

C = System-komprimierte Datei (compressed)

H = Versteckte Datei (hidden)

O = Nicht ständig verfügbare Datei (offline)

R = Nur-lesbare Datei (readonly)

S = System-Datei (system)

T = Nur kurze Zeit verfügbare Datei (temporary)

Beispiel, um ein Listview zu erstellen, wie es oben (siehe Bild) angezeigt wird:

```
ilist&=CreateImageList(3,0)

listview&=CreateListView(%hwnd,%hinstance,0,$0FFFFFFF,-1,$30)
SetImageList(listview&,ilist&)

Dim bereich#,200000
Dim icons#,40000
text$="C:"
bytes&=FilelistToCsv(addr(text$),bereich#,0,0,$1f,icons#,ilist&)
x&=CsvToHeader(listview&,bereich#,addr(y&))
CsvToListview(listview&,(bereich#+y&),(bytes&-y&),x&)
x&=GetLines(listview&)
SetIconsFromMem(listview&,0,icons#,x&)
Dispose icons#
Dispose bereich#

...
ShowListView(listview&,8,40,576,240)
```

### **SetItemText(H,T,S,L)**

Setzt einen neuen Text als Itemtext ein.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
T : Zeiger auf einen String mit dem neuen Text (max. 255 Zeichen)  
S : Long - Index der Spalte von H, in die der neue Text gesetzt werden soll (nullbasierend).  
L : Long - Index der Zeile von H, in die der neue Text gesetzt werden soll (nullbasierend).

Ergebnis: Long - Ungleich 0, bei Fehler 0.

Eine einfache Methode, um einen Itemtext zu ändern / neu zu setzen.

```
text$="Neuer Eintrag"  
SetItemText(listview&,addr(text$),3,6)
```

### **GetItemText(H,B,S,L)**

Liest einen Itemtext aus.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
T : Bereich, in den der ausgelesene Text geschrieben wird  
S : Long - Index der Spalte von H, die ausgelesen werden soll (nullbasierend).  
L : Long - Index der Zeile von H, die ausgelesen werden soll (nullbasierend).

Ergebnis: Long - Ungleich 0, bei Fehler 0.

Eine einfache Methode, um einen Itemtext auszulesen.

```
GetItemText(listview&,bereich#,3,6)  
text$=String$(bereich#,0)
```

### **ExamineColumn(H,S)**

Untersucht, ob eine Spalte Text oder Zahlen beinhaltet.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H, die untersucht werden soll (nullbasierend).

Ergebnis: Long - 1 = Text / 2 = Zahlen.

Eine gute Methode, um z.B. die Sortierart (nach Text oder nach Zahlen) einer unbekannten Dbf oder Csv Datei (eingeladen in ein Listview) einzustellen.

Zuerst ExamineColumn() benutzen, danach [SetColumnSort\(\)](#) (siehe Beispiel 2 für alle Spalten).

Beispiel 1:

```
x&=ExamineColumn(listview&,3)
```

Beispiel 2:

```
Whileloop GetColumns(listview&)  
x&=ExamineColumn(listview&,&loop-1)
```

```
SetColumnSort(listview&,&loop-1,x&)
EndWhile
```

### **SetColumnSort(H,S,F)**

Setzt die Sortier-Methode für eine einzelne Spalte.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H, die untersucht werden soll (nullbasierend).  
F : Long - Flag für die Sortier-Methode

Eine neue Möglichkeit, um für eine einzelne Spalte einzustellen, nach welcher Methode sie sortiert werden soll.  
Mehr hierzu unter [ASortListview\(\)](#).

Für F gelten:

0 = Nicht automatisch sortieren  
1 = Sortieren nach Alphabet  
2 = Sortieren nach Zahlenwert

```
SetColumnSort(listview&,8,1)
```

Beispiel, um die Sortiermethoden aller Spalten eines Listviews automatisch zu setzen:

```
Whileloop GetColumns(listview&)
  x&=ExamineColumn(listview&,&loop-1)
  SetColumnSort(listview&,&loop-1,x&)
Wend
```

### **GetColumnUpdate(H,B)**

Liest Reihenfolge, Breite und Formatierung aller Spalten aus. Wieder setzbar mit SetColumnUpdate().

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen 768 Byte großen Speicherbereich

User können Listviews sehr flexibel an ihre Bedürfnisse anpassen.

Per Drag&Drop können Spalten verschoben werden und die Breite der Spalten kann verändert werden.

Nachteil: Wird ein so verändertes Listview gespeichert (z.B. als Csv-Datei) und später wieder geladen, sind alle Änderungen wieder in den Urzustand zurück gesetzt.

GetColumnUpdate() / SetColumnUpdate() sorgen dafür, das ein Listview so abgespeichert werden kann, das der User seine Einstellungen nur einmal vornehmen braucht, ohne die Listviewdaten (Csv-Datei) selber zu verändern...

B muß ein 768 Byte großer Speicher sein, hierin schreibt GetColumnUpdate() eine "Erinnerungsdatei". Diese kann separat abgespeichert werden oder zusammen mit einer Csv-Datei kombiniert werden, mittels [SetColumnUpdate\(\)](#) können die Einstellungen später wieder auf das Listview angewendet werden.

**Wenn ein User einen oder mehrere Spaltenbutton verschoben hat, dann wird die Reihenfolge der Spalten nicht wirklich verschoben. Die ursprüngliche Reihenfolge bleibt immer erhalten, MS ändert im Listview nur die visuelle Anzeige.**

**Wird so ein verschobenes Listview z.B. ausgedruckt oder editiert wird (usw.), dann arbeiten diese Funktionen mit der internen Spalten-Reihenfolge, nicht mit der visuellen Reihenfolge. Das gilt es zu beachten, ansonsten empfiehlt es sich, den Listview Style \$10 bei [CreateListview\(\)](#) gar nicht erst anzuschalten.**

Beispiel als separate Datei:

```

' *** Csv Datei und Format-Datei laden und anwenden
'
text$="C64.csv"
bytes&=@FileSize(text$)
Dim bereich#,bytes&
ReadFileQuick(addr(text$),bereich#,0,bytes&)
CsvToListview(listview&,bereich#,bytes&,6)
Dispose bereich#

Dim bereich#,768
text$="C64.fmt"
ReadFileQuick(addr(text$),bereich#,0,768)
SetColumnUpdate(listview&,bereich#)
Dispose bereich#

' *** Format-Datei und Csv-Datei speichern
'
Dim bereich#,768
text$="C64.fmt"
GetColumnUpdate(listview&,bereich#)
WriteFileQuick(addr(text$),bereich#,0,768)
Dispose bereich#

text$="C64.csv"
x&=GetNeededMemory(listview&,1)
Dim bereich#,x&
x&=ListviewToCsv(listview&,bereich#,0,0)
WriteFileQuick(addr(text$),bereich#,0,x&)
Dispose bereich#

```

### **SetColumnUpdate(H,B)**

Setzt Reihenfolge, Breite und Formatierung aller Spalten eines Listviews, das vorher mittels `GetColumnUpdate()` ausgelesen wurde.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
 B : Zeiger auf einen 768 Byte großen Speicherbereich

Näheres hierzu unter [GetColumnUpdate\(\)](#).

### **RaiseColumns(H,B,S,G)**

Setzt eine andere Schrift- und Hintergrundfarbe für eine oder mehrere Spalten eines Listviews. Macht Listviews sehr übersichtlich.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
 B : Zeiger auf einen 64 Byte großen Speicherbereich mit Byte-Werten (oder Sonderstring).  
 S : Long - RGB-Wert für Schriftfarbe.  
 G : Long - RGB-Wert für Schrift-Hintergrundfarbe

Hiermit können bestimmte Spalten farbig abgehoben werden, was die Lesbarkeit einer Tabelle stark verbessern kann!

B ist ein 64 Byte großer Speicher, wobei jedes Byte den Index einer Spalte representiert. Steht in einem Byte eine 0, dann wird diese Spalte nicht extra farbig abgehoben. Steht in einem Byte aber eine 1, dann werden in dieser Spalte die Farben S und G verwendet.

B kann auch die Adresse eines Strings sein, wobei Nullen und Einsen als Klartext geschrieben werden. Ist B ein

String, dann können auch weniger als 64 Zeichen uebergeben werden (String muß mit Nullbyte enden, ist bei Profan immer so), ist B ein Speicherbereich, so werden alle 64 Bytes eingetragen, auch wenn das Listview gar keine 64 Spalten besitzt.

Maximal werden 64 Spalten unterstützt (=64 Bytes).

In einem Listview verwenden alle abgehobenen Spalten jedoch immer die gleichen Farben, für jedes Listview können aber andere abgehobene Farben benutzt werden!

Wird in einem Listview [SetBackImage\(\)](#) verwendet, dann werden alle Farbveränderungen von RaiseColumns() aufgehoben (zumindest unter Windows XP). Also aufpassen!!!

Beispiel, um die Spalten 0, 2 und 4 mit schwarzem Text und weißem Hintergrund zu versehen. Alle anderen Spalten bleiben normal.

```
Dim bereich#, 64
Clear bereich#      'bereich# mit Nullen löschen, wichtig!
Byte bereich#, 0=1
Byte bereich#, 2=1
Byte bereich#, 4=1
RaiseColumns(listview&,bereich#, 0, Rgb(255, 255, 255) )
Dispose bereich#
```

Gleiches Beispiel als Stringversion:

```
text$="10101"
RaiseColumns(listview&,addr(text$), 0, Rgb(255, 255, 255) )
```

### **MixRGBs(F1,F2)**

Mischt zwei Farbwerte zu einem.

F1: Long - RGB Farbwert 1  
F2: Long - RGB Farbwert 2

Ergebnis: Long - Mischfarbe (RGB) von F1 und F2.

Eine Funktion, die nichts mit Listviews zu tun hat, aber oft zum Einsatz kommen könnte.

Mischfarben (z.B. zwischen Fenster-Hintergrundsfarbe und Listview-Hintergrundsfarbe) haben den Vorteil, optisch gut zusammen zu passen.

Hier ein Beispiel für RaiseColumns():

```
x&=MixRGBs(GetSysColor(15), $00ffffff)
text$="010101"
RaiseColumns(listview&,addr(text$), $00000000, x&)
```

### **SetBackImage(H,N,F)**

Verpaßt einem Listview eine Hintergrunds-Grafik.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
N : Long - Adresse eines Strings mit dem Namen der Grafikdatei.  
F : Long - Flags, 0 = Bild entfernen / 1 = Bild setzen

Hiermit kann ein Listview eine Hintergrunds-Grafik erhalten.

N ist die Adresse eines Strings. Hier muß der Name der Gafikdatei angegeben werden.

Als Grafikdateien werden unterstützt: BMP,RLE,DIB,JPG,JPE,JPEG,GIF,EMF,WMF,CUR und ICO.

Ist das Bild kleiner als das Listview, dann wird es in an den Seiten reproduziert (Tiles).

Wenn das Listview H gescrollt werden kann, scrollt auch das Bild mit  
Wenn F = 0 ist, wird ein evt. vorher vorhandenes Bild wieder entfernt. Um ein neues Bild zu setzen, muß F = 1 sein.

```
text$="Background.gif"  
SetBackImage(listview&,addr(text$),1)
```

*Achtung! Diese Funktion kann abstürzen, wenn der Internet Explorer 8 Beta (mshtml.dll v8.0.60) installiert ist. In der Release Version 1 hat Microsoft aber den Fehler behoben.*

### **PrintListview(H,W,I,S,L,O,R,U,PS,PZ,US,N,F)**

Bringt ein Listview auf's Papier, wahlweise auch mit Grids, Icons und Checkboxes!

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
W : Long - Handle des Fensters, in dem sich H befindet (%hwnd)  
I : Long - Instance Handle (%hinstance)  
S : Long - Schrifthöhe der auszudruckenden Texte  
L : Long - Größe des linken Papierrands  
O : Long - Größe des oberen Papierrands  
R : Long - Größe des rechten Papierrands  
U : Long - Größe des unteren Papierrands  
PS: Long - Platz zwischen den Spalten  
PZ: Long - Platz zwischen den Zeilen  
US: Zeiger auf einen String mit der Überschrift / Kopfzeile  
N : Zeiger auf einen String mit dem Seitenzahlen-Text oder 0 (voreingestellt = "Seite")  
F : Long - Flags

Eine äußerst mächtige, neue Funktion, dafür auch 13 Parameter "stark".

Natürlich möchte man auch mal ein Listview ausdrucken, jetzt wird das möglich, ohne selber einen großen Aufwand veranstalten zu müssen.

Ausgedruckt werden können alle Texte eines Listviews und wahlweise auch die Gitterlinien, Icons und Checkboxes. Hintergrundgrafiken werden nicht berücksichtigt.

S ist die Schrifthöhe für alle Texte. Werte von 48 bis 64 scheinen hier optimal zu sein.

Hierbei versucht PrintListview() eine Schriftart zu wählen, die der Schriftart innerhalb des Listviews gleicht. Das klappt aber nicht immer, weil der Wert in S kein Pixelwert ist, sondern umgerechnet wird auf die Seitengröße des Druckers, wobei der Wert dann um ein vielfaches höher liegt. Windows kann nicht alle Schriftarten so groß benutzen (z.B. Courier), der Vorteil ist aber, das die Ausdrücke (auch Icons) gestochen scharf werden.

L, O, R und U geben an, wie groß die Seitenränder des Papiers bleiben sollen, hier sind Werte von 200 optimal. PS und PZ geben an, wieviel Platz zwischen den einzelnen Spalten und Zeilen bleiben soll, Werte von 16 ergeben ein gutes Gesamtbild. Experimentieren sie aber ruhig damit...

US muß ein Zeiger auf einen Text sein. Dieser Text wird ganz oben auf Seite 1 gedruckt. Ist US = 0, dann wird kein Titeltex gedruckt.

Der Text in US sollte nicht zu lang sein, da er ansonsten über das Papier hinaus gehen könnte.

N kann ein Zeiger auf einen Text sein, wenn sie als Seitenzahlen-Text etwas anderes als "Seite" benutzen wollen. Ist N = 0, dann wird der zuletzt ausgewählte oder voreingestellte Text verwendet. Der Text in N darf nicht größer sein als 32 Bytes.

F, das sind die Flags (kombinierbar durch OR Verknüpfung oder Addition):

0 = kein Flag gewählt  
1 = Hilfslinien (Grids) werden mit gedruckt, selbst wenn am Bildschirm keine eingestellt sind  
2 = Icons werden mit gedruckt (müssen am Bildschirm aber auch vorhanden sein!)  
4 = Checkboxes werden mit gedruckt (leer), selbst wenn am Bildschirm keine eingestellt sind  
8 = hat der Anwender im Druckermenü "Markierung drucken" eingestellt, dann werden nicht die selektierten Zeilen ausgedruckt, sondern die Zeilen, deren Checkboxes angeklickt sind



16 = führende Leerzeichen (Bytes 32 und 160) in Itemtexten werden nicht automatisch entfernt  
32 = Spalten, die mit RaiseColumn() farbig unterlegt sind, werden beim Drucken grau hinterlegt  
64 = Jede Zeile, die mit RaiseLine() farbig unterlegt wurde, wird beim Drucken grau hinterlegt  
128 = Druckerdialog nicht anzeigen (system-eingestellte Druckerparameter verwenden)

Um zu ermitteln, ob in einem Listview Icons oder Checkboxes aktiv sind, kann [ArelconsPresent\(\)](#) und [AreCheckboxesPresent\(\)](#) benutzt werden.

```
name$="Alle Zeilen ausdrucken (mit Grid, Icons und Checkboxes)."  
PrintListview(listview&,%hwnd,%hinstance,50,200,200,200,200,16,16,addr(name$),0,7)
```

### **AreCheckboxesPresent(H)**

Ermittelt, ob ein Listview mit Checkboxes arbeitet.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Ergebnis: Long - Status.

Ist das Ergebniss 0, dann sind im Listview H keine Checkboxes gesetzt worden.  
Kommt als Ergebniss 1 zurück, dann gibt es in H Checkboxes.

### **ArelconsPresent(H)**

Ermittelt, ob ein Listview mit Icons arbeitet.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Ergebnis: Long - Status.

Ist das Ergebniss 0, dann sind im Listview H keine Icons vorhanden.  
Kommt als Ergebniss 1 zurück, dann gibt es in H Icons.

### **EraseListview(H)**

Entfernt alle Listview-Strukturen eines Listviews innerhalb der DLL. Das ist nötig, um mehr als 64 Listviews pro Programm benutzen zu können.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Bisher waren nur maximal 64 Listviews für jedes Programm erlaubt.  
Das ändert sich mit dieser Funktion, denn hiermit können alle Strukturen innerhalb der Listview.dll wieder geleert werden, was H betrifft. EraseListview() sollte nach jedem DestroyWindow() angewendet werden, bzw. wenn das komplette Fenster gelöscht wurde, in dem sich das Listview befunden hat..  
EraseListview() löscht nicht das Listview selber, sondern nur interne Strukturen.  
Die Anzahl Listviews innerhalb eines Programms ist unbegrenzt, wenn sie diese Funktion benutzen.

### **GetControlParas(B)**

Ermittelt, ob und wo ein Mausklick auf ein Item per linker Maustaste, rechter Maustaste oder linkem Doppel-Mausklick stattgefunden hat.

B : Zeiger auf einen 320 Byte großen Speicherbereich

Ergebnis: Long - 0=kein Itemklick / 1=Doppelklick links / 2=Rechtsklick / 3=Linksklick.

Wenn ein Klick stattgefunden hat dann wird der Speicher B mit Werten gefüllt, mit denen sehr leicht eigene Controls (z.B. Edits oder Comboboxen usw.) an passender Stelle im Listview erzeugt werden können. Eine manuelle und variabelere Methode, um neue Itemtexte einzugeben, ähnlich [EnableEdits\(\)](#).

Ist das Ergebniss 1, 2 oder 3, dann wird B mit folgenden Werten gefüllt (Alles LongInts außer der String):

Offset:

0 = Handle des Listview Controls, in dem der Klick stattfand  
4 = Index der Spalte (nullbasierend), in der der Klick stattfand  
8 = Index der Zeile (nullbasierend), in der der Klick stattfand  
12 = X Position (Pixel) - innerhalb der Listview Abmessungen - des angeklickten Items  
16 = Y Position (Pixel) - innerhalb der Listview Abmessungen - des angeklickten Items  
20 = Breite des angeklickten Items (Pixel)  
24 = Höhe des angeklickten Items (Pixel)  
28 = Font Handle des Listviews  
32 = Textausrichtung innerhalb der angeklickten Spalte (0=links / 1=rechts / 2=zentriert)  
36-63 = noch unbenutzt  
64 = String (kein Zeiger!) mit dem Itemtext (maximal 256 Bytes, abschließend mit Nullbyte)  
-----  
320 Bytes

Damit können sie jetzt an passender Stelle im Listview-Control Edits oder sonstige Controls erstellen, z.B. Edits, die nur Zahlen aufnehmen. Oder bestimmte Spalten sperren, die nicht editiert werden können usw.

Oder ganz einfach nur ermitteln, wo genau der Mausklick stattfand.

Ein Quelltext liegt dem Listview-Paket bei.

Um die Parameter eines bestimmten Items zu erhalten (um z.B. den EnableEdits()-TAB-Effekt nachzubilden) können sie [GetOwnControlParas\(\)](#) benutzen.

In seltenen Fällen kann es beim Gebrauch von GetControlParas() zu ungewollten Scroll-Phänomenen kommen.

Lesen sie in diesem Fall nach unter [ForbidScrollMessages\(\)](#).

Beispiel:

```
Dim bereich#,320
Clear bereich#
y&=GetControlParas(bereich#)
If y<>0
    handle&=Long(bereich#,0)      'Listview Handle
    x&=Long(bereich#,12)          'X Offset
    y&=Long(bereich#,16)          'Y Offset
    b&=Long(bereich#,20)          'Breite
    h&=Long(bereich#,24)          'Höhe
    font&=Long(bereich#,28)       'Font Handle
    text$=String$(bereich#,64)    'Itemtext
    edit&=@CreateEdit(handle&,text$,x&,y&,b&,h&) 'Edit nach den Parametern erstellen
    SetFont edit&,font&
EndIf
Dispose bereich#
```

**GetOwnControlParas(B,H,S,Z)**

Ermittelt einige Daten eines Items, mit denen sehr leicht eigene Controls (z.B. Edits oder Comboboxen usw.) an passender Stelle im Listview erzeugt werden können.

B : Zeiger auf einen 320 Byte großen Speicherbereich  
H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Spalten Index (nullbasierend)  
Z : Long - Zeilen Index (nullbasierend)

Ähnlich [GetControlParas\(\)](#) und darauf aufbauend. Nur können hier gezielt die Daten eines Items ausgelesen werden.

Sehr praktisch, um z.B. Elementsprünge per TAB-Taste vorzunehmen, usw.

Der Speicher B wird in dieser Funktion mit Werten gefüllt, mit denen sehr leicht eigene Controls (z.B. Edits oder Comboboxen usw.) an passender Stelle im Listview erzeugt werden können. Eine manuelle und variable Methode, um neue Itemtexte einzugeben, ähnlich [EnableEdits\(\)](#).

B wird mit folgenden Werten gefüllt (Alles LongInts außer der String):

Offset:  
0 = Handle des Listview Controls, in dem der Klick stattfand  
4 = Index der Spalte (nullbasierend), in der der Klick stattfand  
8 = Index der Zeile (nullbasierend), in der der Klick stattfand  
12 = X Position (Pixel) - innerhalb der Listview Abmessungen - des angeklickten Items  
16 = Y Position (Pixel) - innerhalb der Listview Abmessungen - des angeklickten Items  
20 = Breite des angeklickten Items (Pixel)  
24 = Höhe des angeklickten Items (Pixel)  
28 = Font Handle des Listviews  
32 = Textausrichtung innerhalb der angeklickten Spalte (0=links / 1=rechts / 2=zentriert)  
36-63 = noch unbenutzt  
64 = String (kein Zeiger!) mit dem Itemtext (maximal 256 Bytes, abschließend mit Nullbyte)  
-----  
320 Bytes

Damit können sie jetzt an passender Stelle im Listview-Control Edits oder sonstige Controls erstellen, z.B. Edits, die nur Zahlen aufnehmen. Oder bestimmte Spalten sperren, die nicht editiert werden können usw.

Ein Quelltext liegt dem Listview-Paket bei.

In seltenen Fällen kann es beim Gebrauch von [GetOwnControlParas\(\)](#) zu ungewollten Scroll-Phänomen kommen.

Lesen sie in diesem Fall nach unter [ForbidScrollMessages\(\)](#).

```
Dim bereich#,320
Clear bereich#
GetOwnControlParas(bereich#,listview&,8,6)
x&=Long(bereich#,12)      'X Offset
y&=Long(bereich#,16)      'Y Offset
b&=Long(bereich#,20)      'Breite
h&=Long(bereich#,24)      'Höhe
font&=Long(bereich#,28)   'Font Handle
text$=String$(bereich#,64) 'Itemtext
edit&=@CreateEdit(listview&,text$,x&,y&,b&,h&) 'Edit nach den Parametern erstellen
SetFont edit&,font&
Dispose bereich#
```

## **SetListviewStyle(S)**

Setzt einen Listview-Style, der bei der Erstellung nachfolgender Listviews übernommen wird.

S : Long - Style (siehe Win32.hlp unter List View Window Styles LVS\_)

Einige User möchten gerne erweiterte Einstellmöglichkeiten der Listviews nutzen. Das ging bisher zwar auch mit `GetWindowLong()` / `SetWindowLong()`, so ist es aber einfacher.  
Einfach den gewünschten Listviewstyle eintragen und die nächsten Listviews, die mit `CreateListview()` erstellt werden, bekommen diese(n) Style(s).  
Nur für versierte User...

### **GetRealColumnIndex(H,S)**

Ermittelt den tatsächlichen Index einer Spalte, auch wenn die Spalten per Drag&Drop verschoben wurden.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Spalten Index (nullbasierend)

Ergebnis: Long - Tatsächlicher Spaltenindex, bei Fehler -1

Wenn der User Spalten per Drag&Drop (sofern im Listview gestattet) verschoben hat, dann ändert Windows zwar die visuelle Reihenfolge der Spalten, nicht aber deren Index.  
Das kann dazu führen, das manche Funktionen eine verkehrte Reihenfolge abarbeiten. Mit `GetRealColumnIndex` kann jetzt der richtige Spaltenindex ermittelt werden.

```
GetItemText(listview&,bereich#,GetRealColumnIndex(listview&,4),6)  
text$=String$(bereich#,0)
```

### **CheckIfMarked(H)**

Markiert die Checkboxes von allen Zeilen, die selektiert sind.  
Bei den Zeilen, die nicht markiert sind, wird die Checkbox demarkiert.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Kein Ergebnis.

Natürlich sollten Checkboxes auch aktiviert sein...

Siehe auch [MarkIfChecked\(\)](#).

### **SelectColumnEdits(H,B)**

Bestimmt, welche Spalten vom Anwender editiert werden dürfen, wenn [EnableEdits\(\)](#) benutzt wurde.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen 64 Byte großen Speicherbereich mit Byte-Werten (oder Sonderstring).

Hiermit können bestimmte Spalten vom Anwender editiert werden, oder auch nicht. Voraussetzung ist, das vorher `EnableEdits(handle,1)` aufgerufen wurde.

B ist ein 64 Byte großer Speicher, wobei jedes Byte den Index einer Spalte representiert. Steht in einem Byte eine 0, dann kann diese Spalte nicht editiert werden. Steht in einem Byte aber eine 1, dann kann der Anwender in dieser Spalte Änderungen der Itemtexte durchführen.

B kann auch die Adresse eines Strings sein, wobei Nullen und Einsen als Klartext geschrieben werden. Ist B ein String, dann können auch weniger als 64 Zeichen uebergeben werden (String muß mit Nullbyte enden, ist bei Profan immer so).

Ist B aber ein Speicherbereich, so werden alle 64 Bytes eingetragen, auch wenn das Listview gar keine 64 Spalten besitzt.

Maximal werden 64 Spalten unterstützt (=64 Bytes).

Voreingestellt ist, das durch EnableEdits() alle Spalten editiert werden dürfen.  
Wird in einem Programm EnableEdits() benutzt, aber nicht SelectColumnEdits(), dann kann in allen Spalten editiert werden.

Beispiel, um nur die Spalten 0, 3 und 4 zu editieren zu können. Die anderen Spalten bleiben nicht editierbar.

```
Dim bereich#,64
Clear bereich#      'bereich# mit Nullen löschen, wichtig!
Byte bereich#,0=1
Byte bereich#,3=1
Byte bereich#,4=1
SelectColumnEdits(listview&,bereich#)
Dispose bereich#
```

Gleiches Beispiel als Stringversion:

```
text$="10011"
SelectColumnEdits(listview&,addr(text$))
```

## **GetVar(F)**

Ermittelt den Wert von wichtigen internen Listview.dll-Variablen.

F : Long - Flag

Ergebnis: Long - Flag-bezogener Variablenwert.

Je nach dem, welches Flag in F angegeben ist, wird eine Listview-Variable ermittelt.  
Bisher werden für folgende Flags diese Ergebnisse zurückgegeben:

### **Flag 0:**

Handle des momentan verwendeten nicht manuellen Editcontrols, mit dem der Anwender gerade einen Itemtext editiert. Existiert gerade kein Editcontrol, dann wird 0 zurück gegeben.  
Der Anwender kann nur Editieren, wenn das durch EnableEdits erlaubt wurde.  
Manuelle Edits können mit Flag 6 abgefragt werden.

### **Flag 1:**

Zeiger auf einen 16384 Byte großen Speicherbereich für [RaiseLine\(\)](#).  
Hier steht eine Tabelle mit allen eingefärbten Zeilen, pro Zeile sind 16 Bytes reserviert, für maximal bis zu 1024 Zeilen.  
Pro Zeile sind das also 4 Longint Werte:

```
Offset 0 = Handle des Listviews
      4 = RGB Textfarbe
      8 = RGB Texthintergrundsfarbe
     12 = Nummer der Zeile
```

Sind alle vier Werte 0, dann wurde hier noch keine Zeile eingetragen.

### **Flag 2:**

Index der Spalte (nullbasierend), in der der letzte rechte Mausklick auf einen Spaltenbutton stattfand.  
Diese Abfrage ist komplett selbstgemacht und hat vorerst noch Betastatus. Das originale Listview-Control sieht keine Abfrage eines Rechtsklick auf einen Spaltenbutton vor. Mögliche Anwendungen könnten z.B. das Umbenennen des Spaltenbuttons-Texts oder ein Start/Stopp der automatischen Sortierung sein.  
Wenn als Ergebnis -1 zurückgegeben wird, fand kein Rechtsklick statt.  
Bei jeder Abfrage wird der Status wieder zurückgesetzt auf -1.  
Um zu erfahrend, in welchem Listview der Rechtsklick überhaupt stattfand, kann Flag 3 benutzt werden.

**Flag 3:**

Zusatzinformation zu Flag 2, falls mehrere Listviews verwendet werden.  
Das Ergebnis ist das Handle des Listviews, in dem der Rechtsklick stattfand.

**Flag 4:**

Index der Spalte (nullbasierend), in der der letzte linke Mausklick auf einen Spaltenbutton stattfand.  
Wenn als Ergebnis -1 zurückgegeben wird, fand kein Linksklick statt.  
Bei jeder Abfrage wird der Status wieder zurückgesetzt auf -1.  
Um zu erfahrend, in welchem Listview der Linksklick überhaupt stattfand, kann Flag 5 benutzt werden.

**Flag 5:**

Zusatzinformation zu Flag 4, falls mehrere Listviews verwendet werden.  
Das Ergebnis ist das Handle des Listviews, in dem der Linksklick stattfand.

**Flag 6:**

Handle des momentan verwendeten manuellen Editcontrols, mit dem der Anwender gerade einen Itemtext editiert.  
Existiert dieses Editcontrol nicht (mehr), dann wird 0 zurück gegeben.  
Usergesteuerte Edits können mit Flag 0 abgefragt werden.

**Flag 7:**

Zusatzinformation zu den Usermessages \$1405, \$1406, \$1407 und \$1408, falls mehrere Listviews verwendet werden.  
Das Ergebnis ist das Handle des Listviews, von dem die jeweilige Usermessage gestartet wurde.

**SetIconMode(F)**

Kann Icons transparent oder normal darstellen lassen.

F : Long - Flag, 0 oder 1

Jetzt können Icons auch transparent dargestellt werden.

Windows benutzt solche Icons im allgemeinen, um eine laufende Cut & Paste Aktion darzustellen.

Wird SetIconMode() mit Flag 1 aufgerufen, dann werden alle Icons, die ab jetzt mit [SetIcon\(\)](#), [SetIconsWith\(\)](#) und [SetIconsFromMem\(\)](#) eingebunden werden, transparent dargestellt.

Ist das Flag 0, dann werden die nachfolgenden Icons (wieder) normal dargestellt.

Beispiel (die ersten 10 Icons sind transparent, die restlichen normal):

```
SetIconMode(1)
SetIconsWith(listview&,0,8,10)
SetIconMode(0)
SetIconsWith(listview&,10,8,(GetLines(listview&)-10))
```

**MarkIfChecked(H)**

Selektiert alle Zeilen, deren Checkbox (mit einem Häkchen) markiert ist.

Die Zeilen, deren Checkboxes nicht markiert sind, werden deselektiert.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Kein Ergebnis.

Natürlich sollten Checkboxes auch aktiviert sein....  
Siehe auch [CheckIfMarked\(\)](#).

### **SetFilelistFilter(S)**

Filterfunktion für [FilelistToCsv\(\)](#).

Bis zu 32 Filterstrings können bestimmt werden, die beim Laden einer Dateiliste ignoriert werden.

S : Zeiger auf einen String mit Filter(n) oder 0.

Mit dieser Funktion kann erreicht werden, dass ab dem nächsten FilelistToCsv() eine oder mehrere Dateinamen nicht mit in die Liste übernimmt.

S ist die Adresse eines Strings, der mehrere Strings enthalten kann. Diese werden durch das Zeichen | getrennt werden.

Das Trennzeichen wird durch die Tastenkombination LStrg LAlt < erreicht.

Maximal werden 32 Teilstrings unterstützt. Jeder Teilstring darf höchstens 63 Zeichen groß sein (plus ein Nullbyte).

Groß- und Kleinschrift wird nicht gesondert beachtet, es ist also egal, ob ".bmp" oder ".BMP" angegeben wird.

Wildcards wie \* oder ? werden nicht unterstützt.

Wird SetFilelistFilter() ein zweites Mal oder öfter benutzt, dann werden die vorherigen (Teil) Strings gelöscht.

Ist S = 0, dann wird die vorherige Filtereinstellung gelöscht (kein Filterwort aktiv).

Im gleichen Zusammenhang ist auch noch [SetFilelistNoFilter\(\)](#) interessant.

Beispiel, um keine Bilderdateien in der Fileliste anzuzeigen (jedenfalls die gängigsten):

```
filter$=".bmp|.gif|.jpg|.png"  
SetFilelistFilter(addr (filter$))
```

Alle Dateien, in deren Name einer der Strings ".bmp", ".gif", ".jpg" oder ".png" vorkommen, werden nicht mit in die Fileliste übernommen.

### **SetFilelistNoFilter(S)**

Filterfunktion für [FilelistToCsv\(\)](#).

Bis zu 32 Filterstrings können bestimmt werden, die beim Laden einer Dateiliste angezeigt werden. Alle anderen werden nicht mit in die Liste übernommen.

S : Zeiger auf einen String mit Filter(n) oder 0.

Mit dieser Funktion kann erreicht werden, dass ab dem nächsten FilelistToCsv() nur die Dateinamen mit in die Liste übernommen werden, die in S stehen. Das gilt nur für Programmnamen, nicht für Directories.

S ist die Adresse eines Strings, der mehrere Strings enthalten kann. Diese werden durch das Zeichen | getrennt werden.

Das Trennzeichen wird durch die Tastenkombination LStrg LAlt < erreicht.

Maximal werden 32 Teilstrings unterstützt. Jeder Teilstring darf höchstens 63 Zeichen groß sein (plus ein Nullbyte).

Groß- und Kleinschrift wird nicht gesondert beachtet, es ist also egal, ob ".bmp" oder ".BMP" angegeben wird.

Wildcards wie \* oder ? werden nicht unterstützt.

Wird SetFilelistNoFilter() ein zweites Mal oder öfter benutzt, dann werden die vorherigen (Teil) Strings gelöscht.

Ist S = 0, dann wird die vorherige Filtereinstellung gelöscht (kein Filterwort aktiv).

Im gleichen Zusammenhang ist auch noch [SetFilelistFilter\(\)](#) interessant.

Beispiel, um nur Bilderdateien in der Fileliste anzuzeigen (jedenfalls die gängigsten):

```
filter$=".bmp|.gif|.jpg|.png"  
SetFilelistNoFilter(addr (filter$))
```

Nur Dateien, in deren Name einer der Strings ".bmp", ".gif", ".jpg" oder ".png" vorkommt, werden in die Fileliste übernommen.

Andere Dateien nicht! Seit Version 1.7 sind auch Directories von diesem Filter betroffen.

### **AddItemValues(H,S,B,A)**

Addiert alle Itemtexte einer Spalte als Zahlwerte.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Spalten Index (nullbasierend)  
B : Zeiger auf einen 64 Byte großen Speicherbereich, der den Ergebnisstring aufnimmt.  
A : Long - Anzahl Dezimalstellen (0-16).

Wieder eine Funktion, die dem Anwender jede Menge Arbeit abnimmt.

Addiert werden die Itemtexte (als Zahl) aller Zeilen einer Spalte. Das Ergebnis steht anschließend in B.

Itemtexte, die addiert werden sollen, dürfen nur folgende Zeichen beinhalten:

0 1 2 3 4 5 6 7 8 9 , . - €

Ob als Trennzeichen für die Dezimalstelle ein Punkt (.) oder ein Komma (,) verwendet wird, ist AddItemValues() egal.

Bei der Ausgabe in B wird als Trennzeichen aber immer der Punkt verwendet.

S ist die Nummer der Spalte, deren Werte addiert werden sollen.

B ist der Zeiger auf einen String / Speicherbereich.

A gibt an, wieviele Dezimalstellen bei der Ausgabe verwendet werden. Steht hier z.B. 2, dann wird das Ergebnis auf 2 Stellen abgeschnitten, z.B 15.65.

Würde hier 0 stehen, dann werden nur ganze Zahlen ausgegeben, also 15

Bei A = 16 wäre das Ergebnis 15.6527164916492614.

Ist das Ergebniss in B = 0, dann wird die Null immer ohne Dezimalstellen ausgegeben, also als "0".

AddItemValues rechnet mit REAL10 Zahlen, das sind 80 Bit Fließkommazahlen. Normalerweise rechnen Programme höchstens mit 64 Bit Zahlen. Mit der Listview.dll generierte Listviews können also auch mit riesigen Zahlen operieren!

Die Berechnungen sind noch dazu unglaublich schnell (es wird der FPU Coprozessor verwendet).

Wenn man bedenkt - alle Itemtexte in Listviews sind ja in Wirklichkeit nur Strings - das alle Texte in Zahlen umgerechnet werden müssen, diese addiert werden müssen (als Bruchzahlen mit 80 Bit Genauigkeit) und die Ausgabe wieder in einen String umgerechnet werden muß (Dll's können an das Hauptprogramm nur eine Long-Int Zahl übergeben).

Eine Funktion, große Wirkung.

Beispiel, um die Werte in Spalte 0 zu addieren:

```
Dim bereich#,64
AddItemValues(listview&,0,bereich#,2)
text$=String$(bereich#,0)           'Ausgabezahl als Text (bis 80 Bit)
zahl!=Val(text$)                    'Ausgabezahl als 64 Bit Fließkommazahl
Dispose bereich#
```

Ein schönes Beispiel für Summenbildung in einem Listview ist das Listview.dll-Beispiel "Gebührenverwaltung\_Stammtisch.prf".

### **RaiseLine(H,L,S,G)**

Setzt eine andere Schrift- und Hintergrundfarbe für eine Zeile eines Listviews. Macht Listviews sehr übersichtlich.



H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
L : Long - Zeilen Index (nullbasierend)  
S : Long - RGB-Wert für Schriftfarbe  
G : Long - RGB-Wert für Schrift-Hintergrundfarbe

Hiermit kann eine Zeile Spalten farbige abgehoben werden, was die Lesbarkeit einer Tabelle stark verbessern kann!

L ist die Nummer der Zeile, für die die neuen Farbwerte gelten sollen.

Im Gegensatz zu [RaiseColumns\(\)](#) wird immer nur der Farbwert einer Zeile definiert. Dafür gibt es aber ein Limit von bis zu maximal 1024 andersfarbigen Zeilen gleichzeitig.

S und G sind RGB-Werte einer neuen Farbe. Wird für S und G beidesmal -1 angegeben, dann wird diese Zeile L des Listviews H wieder aus der Tabelle der einzufärbenden Zeilen genommen. Dieser Platz (von 1024) wird also wieder frei.

Wird in einem Listview [SetBackImage\(\)](#) verwendet, dann werden alle Farbveränderungen von sowohl RaiseLine(), als auch RaiseColumns() aufgehoben. Jedenfalls ist das unter Windows XP so. Aufpassen!!!

Interessant zu erwähnen ist auch, dass die Farbgebung von RaiseLine() eine höhere Priorität besitzt, als die Farbgebung von RaiseColumns().

Jedesmal, wenn RaiseLine() ausgeführt wird, muß das Listview neu gezeichnet werden, um die Auswirkung sichtbar zu machen. Will man aber in einem Rutsch mehrere hundert Zeilen einfärben, würden mehrere hundert Listview-Neuzeichnungen zu lange dauern und furchtbar flackern. Deshalb kann man das Neuzeichnen verhindern, indem man zu L den Wert 1.000.000 addiert.

Wer einmal unbedingt wissen muß, welche Zeilen überhaupt welche Farben haben (z.B. um die Färbung mit abzuspeichern), der kann das mittels [GetVar\(1\)](#) herausfinden!

L bleibt immer der tatsächliche Zeilenindex, egal ob noch Zeilen ins Listview eingefügt werden oder ob sich die Reihenfolge der Zeilen geändert hat!

Neu hinzugekommen in Version 1.8 ist das Löschen der gesamten Farbliste mit anschließender Neuzeichnung. Die Löschung aller Farbwerte passiert bei L = -1.

Beispiel, um die Zeilen 100 bis 600 eines Listviews grau zu färben:

```
Whileloop 100,599
  RaiseLine(listview&,( &loop+1000000),0,Rgb(128,128,128))
Endwhile
RaiseLine(listview&,600,0,Rgb(128,128,128))      'Beim letzten Färben Listview auch
updaten!
```

### **GetColumnName(H,B,S)**

Ermittelt den Text eines Spaltenbuttons.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen 256 Byte großen Speicherbereich, in den der Text kopiert wird  
S : Long - Index der Spalte von H, in der ermittelt werden soll (nullbasierend)

Ergebnis: Long - Anzahl Bytes / Buchstaben, die nach B kopiert wurden.

Das ging bisher nur per API.

Hier ein Beispiel, um die Texte aller Spaltenbuttons (eines Listviews) auszulesen:

```
Dim nurso#,256
Whileloop GetColumns(listview&)
  x&=GetColumnName(listview&,nurso#,( &loop-1))
  text$=String$(nurso#,0)
  Print Str$(x&)+" Bytes: "+text$
EndWhile
Dispose nurso#
```

### **SetColumnName(H,B,S)**

Setzt einen neuen Text für einen Spaltenbutton.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen 256 Byte großen Speicherbereich, aus dem der Text kopiert wird  
S : Long - Index der Spalte von H, die gesetzt werden soll (nullbasierend)

Aus das ging bisher nur per API.

Hier ein Beispiel, um die Texte aller Spaltenbuttons (einies Listviews) zu setzen:

```
Dim nurso#,256
text$="neuer Text"
Whileloop GetColumns(listview&)
  String nurso#,0=text$
  x&=SetColumnName(listview&,nurso#,&loop-1)
EndWhile
Dispose nurso#
```

### **SetIconColumn(S)**

Bestimmt, in welcher Spalte das nächste Icon erscheint, das mit [SetIcon\(\)](#), [SetIconFromMem\(\)](#) oder [SetIconWith\(\)](#) ins Listview gesetzt wird. Voreingestellt ist S=0.

S : Long - Index der Spalte (nullbasierend)

Bis Version 1.5 kannte die Listview.dll nur Icons in der linken Spalte. Aufgrund zahlreicher Fürsprachen sind jetzt auch Sub-Icons möglich, also Icons in jeder Spalte.

Es gibt einige Dinge, die hierbei beachtet werden sollten:

- Bei Erstellen eines Listviews mit Sub-Icons mittels [CreateListview\(\)](#) muß Style \$2 gesetzt werden.
- Sub-Icons werden auf dem Papier nicht mit ausgedruckt.

### **GetIcon(H,S,Z)**

Ermittelt die Nummer des Icons, die mit [SetIcon\(\)](#), [SetIconFromMem\(\)](#) oder [SetIconWith\(\)](#) übergeben wurde.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H, in der das Icon ermittelt werden soll (nullbasierend)  
Z : Long - Index der Zeile von H, in der das Icon ermittelt werden soll (nullbasierend)

Ergebnis: Long - Nummer des Icons, unter der es in der Imageliste steht oder -2 wenn hier kein Icon vorhanden ist.

```
GetIcon(listview&,spalte&,zeile&)
```

### **PrintColumns(B)**

Legt fest, welche Spalten eines Listviews beim Drucken mit PrintListview() ausgedruckt werden sollen. Voreingestellt sind alle Spalten.

B : Zeiger auf einen 64 Byte großen Speicherbereich mit Byte-Werten (oder Sonderstring).

Hiermit wird bestimmt, welche Spalten beim nächsten Ausdruck gedruckt werden, und welche nicht. B ist ein 64 Byte großer Speicher, wobei jedes Byte den Index einer Spalte repräsentiert. Steht in einem Byte eine 0, dann wird diese Spalte nicht gedruckt. Steht in einem Byte aber eine 1, dann wird diese Spalte mit ausgedruckt. B kann auch die Adresse eines Strings sein, wobei Nullen und Einsen als Klartext geschrieben werden. Ist B ein String, dann können auch weniger als 64 Zeichen uebergeben werden (String muß mit Nullbyte enden, ist bei Profan immer so), ist B ein Speicherbereich, so werden alle 64 Bytes eingetragen, auch wenn das Listview gar keine 64 Spalten besitzt.

Maximal werden 64 Spalten unterstützt (=64 Bytes).

PrintColumns() bezieht sich immer nur auf [PrintListview\(\)](#), und niemals auf ein bestimmtes Listview!

Beispiel, um die Spalten 0, 2 und 4 beim nächsten PrintListview() auszudrucken, die Spalten 1 und 3 aber nicht.

```
Dim bereich#,64
Clear bereich#      'bereich# mit Nullen löschen, wichtig!
Byte bereich#,0=1
Byte bereich#,2=1
Byte bereich#,4=1
PrintColumns(bereich#)
Dispose bereich#
PrintListview(...)
```

Gleiches Beispiel als Stringversion:

```
text$="10101"
PrintColumns(addr(text$))
PrintListview(...)
```

### **GetItemTextsAsInteger(H,S,B)**

Liest die Itemtexte einer kompletten Spalte aus und speichert die Texte als Integerzahlen.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H, deren Itemtexte ausgelesen werden sollen (nullbasierend)  
B : Zeiger auf einen Speicherbereich, in dem die Texte als Integer gespeichert werden

Ergebnis: Long - Anzahl Integer, die in B gespeichert wurden.

In manchen Fällen ist es sinnvoll, Spalten, in denen Zahlenwerte gespeichert sind, in einem Rutsch auszulesen. GetItemTextsAsInteger() erledigt das für sie, wobei die Funktion alle Itemtexte automatisch in Integerzahlen (vorzeichenbehaftete LongInt's, 32 Bit) umwandelt. Jede Zahl belegt 4 Bytes, alle Zahlen werden nacheinander in B abgelegt.

B muß ausreichend dimensioniert sein, die Anzahl zu belegender Bytes berechnet sich nach:

Anzahl Zeilen in H \* 4

Die Anzahl Zeilen in H kann via [GetLines\(\)](#) ermittelt werden.

Eine mögliche Anwendung für GetItemTextsAsInteger() wären z.B. Linien- oder Balkendiagramme.

In diesem Zusammenhang ist sicher auch noch [GetEdgeIntegers\(\)](#) interessant.

Möchte sie die Itemtexte als Fließkommazahlen (Dezimalzahlen, Bruchzahlen) ausgelesen bekommen, dann benutzen sie bitte [GetItemTextsAsFloat\(\)](#).

Beispiel:

```
Dim bereich#,GetLines(listview&)*4
x&=GetItemTextsAsInteger(listview&,3,bereich#)
```

```
Whileloop x&
  Print Long(bereich#, (&loop-1)*4)
Wend
Dispose bereich#
```

### **GetEdgeIntegers(B,A,L,H)**

Ermittelt aus einer Tabelle mit Integerzahlen den niedrigsten und den höchsten Wert.

B : Zeiger auf einen Speicherbereich mit Integerzahlen  
 A : Long - Anzahl Integerzahlen in B  
 L : Zeiger auf eine 32 Bit Variable, um den niedrigsten Wert zu empfangen  
 H : Zeiger auf eine 32 Bit Variable, um den höchsten Wert zu empfangen

Eine schnelle Methode, um den niedrigsten und höchsten Wert aus einer Tabelle mit vielen Werten zu ermitteln.  
 Die Werte müssen nacheinander in B vorliegen, wobei jeder Wert ein vorzeichenbehaftetes LongInt ist, 32 Bit groß.

```
Dim bereich#,28
Long bereich#,0=3546
Long bereich#,4=18506
Long bereich#,8=-4623
Long bereich#,12=-11003
Long bereich#,16=18
Long bereich#,20=0
Long bereich#,24=-467
GetEdgeIntegers(bereich#,7,addr(x&),addr(y&))
Print x&
Print y&
Dispose bereich#
```

**-11003**  
**18506**

### **GetItemTextsAsFloat(H,S,B)**

Liest die Itemtexte einer kompletten Spalte aus und speichert die Texte als Fließkommazahlen.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
 S : Long - Index der Spalte von H, deren Itemtexte ausgelesen werden sollen (nullbasierend)  
 B : Zeiger auf einen Speicherbereich, in dem die Texte als Floats (64 Bit) gespeichert werden

Ergebnis: Long - Anzahl Floats, die in B gespeichert wurden.

In manchen Fällen ist es sinnvoll, Spalten, in denen Zahlenwerte gespeichert sind, in einem Rutsch auszulesen. GetItemTextsAsFloat() erledigt das für sie, wobei die Funktion alle Itemtexte automatisch in Fließkommazahlen (vorzeichenbehaftet, 64 Bit) umwandelt. Jede Zahl belegt 8 Bytes, alle Zahlen werden nacheinander in B abgelegt. B muß ausreichend dimensioniert sein, die Anzahl zu belegender Bytes berechnet sich nach:

Anzahl Zeilen in H \* 8

Die Anzahl Zeilen in H kann via [GetLines\(\)](#) ermittelt werden.

Eine mögliche Anwendung für GetItemTextsAsFloat() wären z.B. Linien- oder Balkendiagramme. In diesem Zusammenhang ist sicher auch noch [GetEdgeFloats\(\)](#) interessant.

Beispiel:

```
Dim bereich#,GetLines(listview&)*8
x&=GetItemTextsAsFloat(listview&,3,bereich#)
Whileloop x&
  GetFloat(itemtexts#,&loop-1)*8,addr(float!))
  Print float!
Wend
Dispose bereich#
```

### **GetEdgeFloats(B,A,L,H)**

Ermittelt aus einer Tabelle mit Fließkommazahlen den niedrigsten und den höchsten Wert.

B : Zeiger auf einen Speicherbereich mit Integerzahlen  
A : Long - Anzahl Floats in B  
L : Zeiger auf eine 64 Bit Fließkomma-Variable, um den niedrigsten Wert zu empfangen  
H : Zeiger auf eine 64 Bit Fließkomma-Variable, um den höchsten Wert zu empfangen

Eine relativ schnelle Methode, um den niedrigsten und höchsten Wert aus einer Tabelle mit vielen Werten zu ermitteln.

Die Werte müssen nacheinander in B vorliegen, wobei jeder Wert eine vorzeichenbehaftete 64 Bit Fließkommazahl sein muß.

```
GetEdgeFloats(bereich#,7,addr(lofloat!),addr(hifloat!))
Print lofloat!
Print hifloat!
```

### **GetFloat(B,O,F)**

Liest eine 64 Bit Fließkommazahl (FLOAT8) aus einem Speicherbereich aus.

B : Zeiger auf einen Speicherbereich mit Floats  
A : Long - Offset innerhalb B  
F : Zeiger auf eine 64 Bit Fließkomma-Variable, um die Zahl zu empfangen

Eine einfache Möglichkeit, eine Fließkommazahl aus einem Speicher auszulesen.

64 Bit Floats sind immer 8 Bytes groß.

```
GetFloat(bereich#,0,addr(float!))
Print float!
```

### **DeleteDoubleItems(H,S)**

Löscht alle doppelt oder mehrfach vorkommen Itemtexte in einer Spalte eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten ListView Controls  
S : Long - Index der Spalte von H, deren mehrfach vorkommende Itemtexte gelöscht werden sollen (nullbasierend)

Eine schnelle und bequeme Methode, doppelte, bzw. mehrfache Einträge zu löschen.

Um ein komplettes ListView damit zu bearbeiten, müssen alle Spalten durchlaufen werden, z.B. so:

```
SendMessage(listview&,11,0,0) 'Neuaufbau des Listviews verhindern!
```

```
Whileloop GetColumns(listview&)
    DeleteDoubleItems(listview&,( &loop-1))
EndWhile

SendMessage(listview&,11,1,0)      'Wiederaufbau des Listviews (standart).
```

### **GetAllCheckboxStates(H,B)**

Ermittelt den Status aller Checkboxes-Markierungen eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
 B : Zeiger auf einen Speicherbereich, der den Status als Bytefolge aufnimmt.

Ergebnis: Long - Anzahl geschriebener Bytes in B.

Ermittelt den Status aller Checkboxes innerhalb eines Listviews. Natürlich nur, wenn Checkboxes vorhanden sind...

Um Checkboxes in Listviews verwenden zu können, muß Style 4 der LVS\_EX\_STYLES gesetzt werden. Mehr Infos hierzu unter [CreateListview\(\)](#).

Das Ergebnis ist eine Folge von Nullen und Einsen, die in B als Text gespeichert wird und von dort aus weiter bearbeitet oder gespeichert werden kann. Eine 0 bedeutet, an dieser Stelle ist die Checkbox nicht markiert. Eine 1 steht für eine markierte Checkbox.

Verwandte Themen: [SetAllCheckboxStates\(\)](#), [GetCheckboyState\(\)](#), [SetCheckboxState\(\)](#), [GetChecked\(\)](#).

### **SetAllCheckboxStates(H,B)**

Setzt den Status aller Checkboxes-Markierungen eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
 B : Zeiger auf einen Speicherbereich, der den Status als Bytefolge vorgibt.

Ergebnis: Long - Anzahl gelesener Bytes in B.

Setzt den Status aller Checkboxes innerhalb eines Listviews. Natürlich nur, wenn Checkboxes vorhanden sind...

Um Checkboxes in Listviews verwenden zu können, muß Style 4 der LVS\_EX\_STYLES gesetzt werden. Mehr Infos hierzu unter [CreateListview\(\)](#).

Eine Folge von Nullen und Einsen, die in B als Text gespeichert ist, wird als Grundlage der Checkbox Markierungen genommen. Eine 0 bedeutet, an dieser Stelle ist die Checkbox nicht markiert. Eine 1 steht für eine markierte Checkbox.

Der Speicher in B kann von Hand erstellt werden, oder durch die Funktion [GetAllCheckboxStates\(\)](#).

```
text$="001100100100100010110110011"
SetAllCheckboxStates(listview&,Addr(text$))
```

Verwandte Themen: [GetAllCheckboxStates\(\)](#), [GetCheckboyState\(\)](#), [SetCheckboxState\(\)](#), [GetChecked\(\)](#).

### **GetDllVersion()**

Ermittelt die vorliegende Listview.dll Version.

Ergebniss: Long - Aktuelle Versioninfo

Das Ergebnis ist immer eine vierstellige Zahl!

Wird beispielsweise 1702 ausgegeben, dann bedeutet das: Version 1.7 Release 0.2, also 1.7.0.2.

### **SetLineNumbers(H,A,S)**

Nummeriert die Items einer Spalte.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
A : Long - Ausgangsnummer, mit der die Nummerierung startet  
S : Long - Index der Spalte von H, die nummeriert werden soll (nullbasierend)

Eine schnelle und bequeme Methode, ein Listview zu nummerieren. Stand vorher schon ein Text in einem Item, dann wird er ersetzt.

```
SetLineNumbers(listview&,1,0)
```

### **SetColumnAlignment(H,S,A)**

Nachträgliche Textausrichtung einer Spalte.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H, die ausgerichtet werden soll (nullbasierend)  
A : LONG - Ausrichtung

Hiermit kann eine Spalte nachträglich ausgerichtet werden. Werte für A sind:

0 = Linksbündig  
1 = Rechtsbündig  
2 = Zentriert

```
SetColumnAlignment(listview&,spalte&,0)  
Update(listview&)
```

### **EnableDragDrop(H,F)**

Erlaubt in einem Listview Drag & Drop Aktionen (oder auch nicht). Also das Verschieben von Items mittels der Maus.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
F : Long - Flag

Eine der mächtigsten Funktionen der Listview.dll!

Mit EnableDragDrop() können Verschiebungen von Texten per Maus eingeschaltet (oder wieder abgeschaltet) werden, und zwar für jedes einzelne Listview.

H ist das Listview, dem D&D zugeschaltet (bzw. weggeschaltet) werden soll. Ob und wie ein Listview Drag&Drop unterstützen soll, bestimmt das Flag F:

0 = H soll kein Drag&Drop unterstützen  
1 = Drag & Drop für H einschalten  
2 = H unterstützt nur Drag&Drop in/von Listviews, die mittels der Listview.dll erstellt wurden (no extern)  
4 = H unterstützt nur Drag&Drop innerhalb sich selber (Drag&Drop-Sortierung)  
8 = Bei Verschiebungen aus H wird die Quellzeile aus dem Listview entfernt (Move)  
16 = Subitem-Modus. Es werden keine Zeilen verschoben, sondern einzelne Itemtexte vertauscht

32 = Nur zusammen mit Flag 16. Schaltet im Subitem-Modus den Move-Modus an  
64 = Nach jeder DragDrop-Aktion abschliessende Leerzeilen automatisch entfernen  
128 = Nicht zusammen mit Flag 16. Die verschobenen Zeilen immer unten anhängen  
256 = Drag&Drop nicht ausführen, sondern nur wichtige Daten dazu an den Programmierer übergeben  
512 = Nur zusammen mit Flag 256! Das Zielcontrol ist kein Listview, sondern ein Treeview!

Alle Flags gelten lokal, sie können also für jedes Listview unterschiedlich eingestellt werden.

#### Anmerkungen:

Wird ein Eintrag von einem Listview in ein anderes Listview geschoben, und beide Controls wurden von der Listview.dll erzeugt, dann wandert die neue Textzeile oberhalb der Zeile, auf die es gezogen wurde. Das funktioniert auch mit mehreren Zeilen gleichzeitig. Aber wird versucht, mehrere Zeilen gleichzeitig in ein und selbe Listview zu verschieben, dann wird nur der oberste selektierte Eintrag verschoben. Ist Flag 128 gesetzt, wandern die neuen Zeilen immer ans Ende der Liste.

Wurde aus einem externen Control (z.B. Windows-Explorer) in ein Listview.dll-Listview gezogen, dann wandert die neue Zeile, bzw. die neuen Zeilen immer an das Ende der Liste.

Mit Flag 2 bin ich noch sehr unzufrieden, weil Windows hier anscheinend Fehler beim Abbrechen eines gültigen Drag&Drop Vorganges macht. Wird bei gesetztem Flag 2 trotzdem ein Eintrag in ein externes Control verschoben, kann es passieren, dass das externe Programm eine OLE-Meldung ausgibt. Gedroppt wird aber trotzdem nichts. Anders herum ändert sich beim Ziehen von einem externen Control zu einem Dll-Listview der Mauscursor zu einem Kreuz-Symbol. Aber auch hier wird ordnungsgemäss nicht gedroppt.

Flag 16 ist ein echtes Schmäckerl! Hier reagiert das Drag&Drop vollständig anders. Es werden keine Zeilen hin und her geschoben, sondern nur einzelne Itemtexte einer Spalte. Hierbei werden die beiden betroffenen Texte vertauscht.

Wird der Text in den ungültigen Randbereich eines Listviews gezogen, dann wird er gelöscht.

Um im Flag-16-Modus Einträge in ein anderes (Dll-erzeugtes) Listview zu verschieben, muss das Flag bei beiden Listviews gesetzt sein.

Wird zusätzlich zum Flag 16 das Flag 32 gesetzt, dann werden die beiden Texte nicht nur vertauscht, sondern der Quelltext auch wirklich aus der Spalte entfernt und der Zieltext zusätzlich in die neue Spalte eingefügt. Das entspricht natürlich nicht dem üblichen Abhängigkeits-Verhältniss von Spalten und Zeilen eines Listviews, mag aber in bestimmten Programmen durchaus sinnvoll sein. Im Modus 32 werden evt. vorhandene Icons und Checkboxes nicht mitverschoben, da sich diese nicht zwangsläufig nur auf eine Spalte beziehen, sondern auf die gesamte Zeile! Vermeiden sie diese darum im 32er Modus.

Wird der Text im 32er Modus in einen nicht gültigen Bereich des Listviews verschoben, wird die Drag&Drop Aktion nicht durchgeführt.

Ist Modus 32 gesetzt, dann wird Flag 8 ignoriert.

Ist zusätzlich zu den Flags 16 und 32 noch das Flag 64 gesetzt, dann werden abschliessende Leerzeilen automatisch entfernt. Es können durch die Verschiebungen nämlich solche entstehen und nicht immer sind diese erwünscht.

Ist Flag 16 gesetzt, können nicht mehrere selektierte Texte gleichzeitig verschoben werden. Es wird in diesem Fall nur der Eintrag unter dem Mauscursor verschoben.

Ist Flag 256 gesetzt, werden keine Einträge verschoben. Stattdessen meldet jetzt die Funktion [GetDragDropParas\(\)](#) dem Programmierer aber, dass eine D&D-Aktion stattfand, zwischen welchen Controls und wohin genau. Der Programmierer kann jetzt selber darauf reagieren. Neu ist Flag 512, der als Zielcontrol ein Treeview zurück meldet. Siehe auch [GetDragDropParas\(\)](#).

#### **DeleteSpaceLines(H,F)**

Entfernt (abschliessende) Leerzeilen aus einem Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
F : Long - Flag

Hiermit können alle Leerzeilen aus einem Listview entfernt werden, oder aber nur abschliessende Leerzeilen (von unten nach oben alle Leerzeilen entfernen, bis eine Zeile gefunden wurde, die Text enthält). Wie verfahren werden soll, bestimmt das Flag F:



0 = Alle entfernen  
1 = Nur abschliessende Leerzeilen entfernen

### **GetDragDropParas(B)**

Ermittelt, ob und wo eine Drag&Drop Aktion stattfand. Hierzu benötigt das Control aber ein gesetztes Flag 256 bei [EnableDragDrop\(\)](#).

B : Zeiger auf einen 320 Byte großen Speicherbereich

Ergebnis: Long - 0=keine neue Drag&Drop-Aktion / 1=Drag&Drop-Aktion fand statt.

Wenn der User eine Drag&Drop-Aktion durchgeführt hat, dann wird der Speicher B mit Werten gefüllt, mit denen sehr leicht selber auf die Aktion reagiert werden kann. Funktioniert nur mit Aktionen, die zwischen zwei Listview.dll's ausgingen. Externes Drag&Drop wird nicht unterstützt.

Ist das Ergebniss 1, dann wird B mit folgenden Werten gefüllt (Alles LongInts):

Offset:  
0 = Handle des Listview Controls, aus der eine Zeile/ein Text gezogen wurde (Quelle)  
4 = Index der Spalte (nullbasierend), aus der gezogen wurde  
8 = Index der Zeile (nullbasierend), aus der gezogen wurde  
12 = Handle des Listview Controls, in den eine Zeile/ein Text gezogen wurde (Ziel)  
16 = Index der Spalte (nullbasierend), in die gezogen wurde  
20 = Index der Zeile (nullbasierend), in die gezogen wurde  
24-319 = noch unbenutzt  
-----  
320 Bytes

Damit können sie jetzt an passender Stelle im Listview-Control selber auf die Drag&Drop-Funktion reagieren, so wie sie es wünschen.

Wurde Flag 512 bei EnableDragDrop() gesetzt, dann steht an Offset 12 das Treeviewhandle, an Offset 16 das Handle des angeklickten Items und an Offset 20 steht das TVHT-Flag (Näheres zu TVHT\_ siehe Windows-Literatur).

Beispiel (Messageloop):

```
Dim bereich#,321
While 1

    waitinput
    Case %key=2:BREAK

    x&=GetDragDropParas(bereich#)
    If x&=1
        handle1&=Long(bereich#,0)
        x1&=Long(bereich#,4)
        y1&=Long(bereich#,8)
        handle2&=Long(bereich#,12)
        x2&=Long(bereich#,16)
        y2&=Long(bereich#,20)
    EndIf

Wend
Dispose bereich#
```

## **ConvertDatas(H,S,F)**

Convertiert alle Dezimalzahlen einer Spalte in das deutsche oder amerikanische Zahlenformat und fügt wahlweise das Eurozeichen (€) dazu.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H (nullbasierend)  
F : Long - Flag

Hiermit werden alle nicht realen Zahlen einer Spalte so umgewandelt, das sie anstelle des Punktes ein Komma als Trennzeichen erhalten.

Ist F=1, dann wird zusätzlich ein Leerzeichen und das Eurozeichen angefügt.

Wenn F=2 ist, dann werden alle Kommata gegen Punkte ausgetauscht, also die amerikanische Schreibweise verwendet.

Die Flags sind kombinierbar.

Beispiel für die Verwendung von F:

```
Flag = 0: Aus 6.53 wird 6,53  
Flag = 1: Aus 6.53 wird 6,53 €  
Flag = 2: Aus 6,53 wird 6.53  
Flag = 3: Aus 6,53 wird 6.53 €
```

Beispiel, um alle Items eines Listview zu convertieren und mit Eurozeichen zu versehen:

```
Whileloop GetColumns(listview&)  
  ConvertDatas(listview&, (&loop-1), 1)  
EndWhile
```

Verwandte Themen: [AddItemValues\(\)](#), [ExamineColumn\(\)](#).

## **ForbidScrollMessage(F)**

Verhindert das automatische Scrollen von nur zum Teil sichtbaren Items.

F : Long - Flag

An einigen Stellen verwendet die Listview.dll eine Message, welche nur teilweise sichtbare Einträge komplett in das Listview einschiebt.

Das kann aber hin und wieder zu ungewollten Verschiebungen führen, nämlich immer dann, wenn in der Message-Schleife nicht Waitinput benutzt wird, oder der MessageMode 2 verwendet wird. Betroffen hiervon sind dann die Funktionen [SelectLine\(\)](#), [GetControlParas\(\)](#) und [GetOwnControlParas\(\)](#).

ForbidScrollMessage() ist dazu gedacht, das automatische Sichtscrolling kurzfristig auszuschalten, wenn diese Funktionen im Programm benutzt werden und die Message-Schleife ungewollte Notify-Messages durchlässt. Kurzfristig deswegen, weil die betroffene Message nämlich auch im Edit- und Drag&Drop Modus benutzt wird und im ausgeschalteten Zustand ansonsten sehr unerwünschte Effekte produzieren wird!

F=0 ist der Normalzustand. Ist F=1, dann ist das Autoscrolling deaktiviert.

```
ForbidScrollMessage(1)  
y&=Getcontrolparas(bereich#)  
ForbidScrollMessage(0)  
If y&<>0  
...
```

## **ExchangeBytes(H,S,Q,Z)**

Tauscht einzelne Bytes innerhalb einer Listview-Spalte gegen andere aus.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Index der Spalte von H (nullbasierend)  
Q : Long - ASCII Code der Zeichen, die ausgetauscht werden sollen (Quelle).  
Z : Long - ASCII Code der Zeichen, mit denen alle Q-Zeichen überschrieben werden (Ziel).

Eine sinnvolle Ergänzung zu [ExchangeSeparator\(\)](#) und diversen Dateien. ExchangeBytes() kann z.B. nachträglich die Trennzeichen einer Csv-(ähnlichen) Datei ändern. Wenn sich die Daten schon im Listview befinden. Ein Beispiel wäre vielleicht, den Quellcode einer Profandatei in ein Listview einzulesen. Da sich hierin aber häufig Kommata und Semikolons (neben der Erkennung einer Zeile übliche Trennzeichen) befinden, würde so eine Datei immer verkehrt eingelesen werden, und es entstünden mehr Zeilen, als gewünscht. Hier müssten jetzt vor dem Einlesen der Datei alle Anführungszeichen, Kommata und Semikolons umgetauscht werden in quasi nie verwendete Zeichen. Dann wird die Datei eingelesen und danach werden die drei Zeichentypen wieder zurück verwandelt.

So sieht der Code dazu aus (mehr dazu im Democode LV\_Quellcodes\_einlesen.prf):

```
text$="Dateiname.prf"
bytes&=@FileSize(text$)
If bytes&>0
    Dim bereich#,bytes&+1
    Clear bereich#
    ReadFileQuick(addr(text$),bereich#,0,bytes&)

    ExchangeSeparator(bereich#,bytes&,10,11,1) 'Leerzeilen erzwingen

    ExchangeSeparator(bereich#,bytes&,@Ord(",") ,@Ord(" ") ,1)
    ExchangeSeparator(bereich#,bytes&,@Ord(";") ,@Ord("æ") ,1)
    ExchangeSeparator(bereich#,bytes&,34,@Ord("P") ,1)

    CsvToListview(listview&,bereich#,bytes&,1)

    ExchangeBytes(listview&,0,@Ord(" ") ,@Ord(",") )
    ExchangeBytes(listview&,0,@Ord("æ") ,@Ord(";") )
    ExchangeBytes(listview&,0,@Ord("P") ,34)

    Dispose bereich#
EndIf
```

### **SetPrintAttributes(F,P1,P2,P3,P4)**

Zusätzliche Features für Listview.dll's [PrintListview\(\)](#) an- oder abschalten.

F : Long - Flag  
P1: Long - Parameter 1  
P2: Long - Parameter 2  
P3: Long - Parameter 3  
P4: Long - Parameter 4

SetPrintAttributes() setzt zusätzliche Attribute für den/die nächsten Druckaufträge via PrintListview(). Ist Flag = 0, dann kann eine Fusszeile angegeben werden, die unter jeder ausgedruckten Seite erscheint. Diese Zeile darf maximal 256 Bytes lang sein, bitte darauf achten! Die Fusszeile wird bei P1 angegeben, als Zeiger auf einen String oder Bereich. Ist P1 = 0, dann wird keine Fusszeile (mehr) ausgegeben.

Beispiel:

```
fusszeile$="Dieser Text ist eine Fusszeile..."
SetPrintAttributes(0,addr(fusszeile$),0,0,0)
PrintListview(...)
```

Bisher gibt es nur Flag 0.

### **SetLineHeight(H,Y)**

Setzt manuell eine neue Höhe für alle Listview-Zeilen.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
Y : Long - Höhe in Pixel

Ergebnis: Long - Zeiger auf eine Imageliste.

Normalerweise passt das Listview seine Zeilenhöhe automatisch dem verwendeten Zeichensatz (Font) an. Eine Message dazu ist von Microsoft nicht vorgesehen.

Trotzdem gibt es einen Trick, um dennoch die Höhe seiner Listview-Zeilen zu bestimmen. Dazu generiert SetLineHeight() eine Imageliste mit unsichtbaren, 1 Pixel breiten und Y Pixel hohen Icons, und weist diese dem Listview zu. Das klappt wirklich gut.

Zu beachten ist natürlich, dass eine eventuell vorher dem Listview zugewiesene Imageliste dadurch deaktiviert wird. Wenn das Listview also vorher Icons anzeigt, ist SetLineHeight() natürlich ungeeignet, weil sonst die Icons verschwinden.

Weiterhin sollte die mit SetLineHeight() erstellte Imageliste am Ende wieder freigegeben werden, um die Ressourcen zu schonen. Das Freigeben funktioniert mit DestroyImageList(), wenn sie Listview's Datei Listview\_Funktionen.inc eingebunden haben. Beispiel:

```
freelist=&SetLineHeight(listview&,80)
...
DestroyImageList(freelist&)
```

### **GetLastKey(L,T)**

Liest den letzten Tastendruck aus, der in einem Listview stattfand.

L : Zeiger auf eine 32 Bit Long-Int-Variable, um den Handle des Listviews zu empfangen, in dem der letzte Tastendruck auftrat.

T : Zeiger auf eine 32 Bit Long-Int-Variable, um den Scancode des letzten Tastendrucks zu erfahren.

Ergebnis: Long - 0=kein Tastendruck erfolgt / 1=ein Tastendruck fand seit der letzten Abfrage statt.

Eine einfache Möglichkeit, einen Tastendruck in einem Listview zu ermitteln. Allerdings werden nur die Tasten gemeldet, die ein Listview auch an das Parentfenster sendet. Eine Möglichkeit, alle Tastendrucke zu erfahren, ist die Usermessage [\\$1400](#). Darum ist die Message in jedem Fall vorzuziehen.

```
x=&GetLastKey(Addr(lhandle&),Addr(taste&))
If x&
  Print "Taste mit dem Scancode "+Str$(taste&)+" fand statt in Listview "+Str$(lhandle&)+". "
EndIf
```

### **SetColumnsWidthLimits(L,H)**

Schränkt den User ein, wie klein oder gross er die Spalten eines Listviews mit der Maus ziehen kann.

L : Minimumwert in Pixel.  
H : Maximumwert in Pixel.

Der Benutzer von Listviews kann mit der Maus die Breite seiner Spalten variabel gestalten, sofern das passende Flag dafür gesetzt ist. Aber nicht immer ist das vom Programmierer so gewünscht. Darum kann dieser mittels `SetColumnsWidthLimits()` die Breite einschränken, sowohl die grösst-mögliche Breite, als auch die kleinstmögliche. Wird `SetColumnsWidthLimits()` nicht benutzt, dann beträgt der Minimalwert einer Spalte Null, und der Maximalwert 32768 Pixel.

Die Funktion verändert nicht die aktuelle Breite der Spalten!

```
SetColumnsWidthLimits(20,320)
```

### **ListviewToRaw(H,B)**

Auslesen eines Listviews in den Speicher im RAW-Format.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
B : Zeiger auf einen Bereich, in den alle Listview-Texte als Rohdaten gespeichert werden.

Ergebnis: Long - Anzahl Bytes, die in den Bereich B geschrieben wurden.

Eine weitere Funktion, um den Inhalt eines Listviews abzuspeichern. Im Gegensatz zu [ListviewToCsv\(\)](#) und [ListviewToDbf\(\)](#) schreibt diese Funktion den Inhalt des Listviews H aber nicht in einer bekannten Datei-Struktur, sondern nutzt ein eigenes Format, welches die Nachteile von CSV und DBF kompensiert. Dieses RAW-Format ähnelt dem CSV-Format sehr, jedoch benutzt es als Trennzeichen der einzelnen Spalten das Byte 2 und als Erkennungszeichen für eine neue Zeile das Byte 3.

Der Speicher B muß natürlich groß genug dimensioniert sein. Wie groß er mindestens sein muß, kann zuvor mit [GetNeededMemory\(\)](#) (Flag 1) ermittelt werden. Mit [WriteFileQuick\(\)](#) läßt er sich dann auf einem Datenträger abspeichern.

Ein mit `ListviewToRaw()` gespeicherter Listviewinhalt kann mit [RawToListview\(\)](#) wieder eingelesen werden.

```
ListviewToRaw(listview&,bereich#)
```

### **RawToListview(H,B)**

Einlesen eines Speichers im RAW-Format in ein Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
B : Zeiger auf einen Bereich, in dem Listview-Texte als Rohdaten abgelegt sind.

Liest Daten in ein Listview ein, die mittels [ListviewToRaw\(\)](#) gespeichert wurden. Dafür müssen schon Spalten in passendem Umfang vorhanden sein. Neue Zeilen werden dagegen automatisch erzeugt. Um die neuen Zeilen an anderer Stelle als unten einzufügen, ist [SetIndex\(\)](#) zu benutzen.

Das RAW-Format ähnelt dem CSV-Format sehr, jedoch benutzt es als Trennzeichen der einzelnen Spalten das Byte 2 und als Erkennungszeichen für eine neue Zeile das Byte 3.

```
RawToListview(listview&,bereich#)
```

### **SetItemTextEx(H,T,S,L,A)**

Setzt einen neuen Text als Itemtext ein, der auch länger sein darf als 255 Zeichen.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
T : Zeiger auf einen String oder Bereich mit dem neuen Text.  
S : Long - Index der Spalte von H, in die der neue Text gesetzt werden soll (nullbasierend).  
L : Long - Index der Zeile von H, in die der neue Text gesetzt werden soll (nullbasierend).  
A : Long - Anzahl Zeichen des Texts in T.

Erweiterung der Funktion [SetItemText\(\)](#). Eine einfache Methode, um einen Itemtext zu ändern / neu zu setzen.

```
text$="Neuer Eintrag"  
SetItemText(listview&,addr(text$),3,6,len(text$))
```

### **GetItemTextEx(H,B,S,L,A)**

Liest einen Itemtext aus, der auch länger sein darf als 255 Zeichen.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
T : Bereich, in den der ausgelesene Text geschrieben wird  
S : Long - Index der Spalte von H, die ausgelesen werden soll (nullbasierend).  
L : Long - Index der Zeile von H, die ausgelesen werden soll (nullbasierend).  
A : Long - Anzahl verfügbarer Bytes in T.

Erweiterung der Funktion [GetItemText\(\)](#). Eine einfache Methode, um einen Itemtext auszulesen.

```
Dim bereich,32768  
GetItemTextEx(listview&,bereich#,3,6,32768)  
text$=String$(bereich#,0)  
Dispose bereich#
```

### **BuildListview(H,S,L,B,T)**

Erzeugt leere Spalten und Zeilen in einem Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
S : Long - Anzahl Spalten, die in H erzeugt werden sollen.  
L : Long - Anzahl Zeilen, die in H erzeugt werden sollen.  
B : Long - Breite der erzeugten Spalten.  
T : Long - Textformatierungsflag.

Eine superschnelle Methode, um das Listview H mit Spalten und Zeilen zu füllen.

B gibt an, wie viele Pixel breit die erzeugten Spalten werden sollen.

Mit F kann die Textausrichtung der zeugten Items in dieser Spalte bestimmt werden. Sichtbar wird das aber erst, wenn Texte hinein gesetzt werden. Flags für F können sein:

0 = linksbündig  
1 = rechtsbündig  
2 = zentriert

Zunächst wird das Listview komplett gelöscht, sollte es noch Spalten oder Zeilen mit Texten enthalten. Also Vorsicht!

Dann wird das Listview neu aufgebaut, wobei zuerst S leere Spalten erzeugt werden. Danach L leere Zeilen. Um so ein "nacktes" Listview zu bestücken, empfehlen sich die Funktionen [SetColumnName\(\)](#), [SetItemText\(\)](#) und [SetItemTextEx\(\)](#).

```
BuildListview(listview&,6,30,80,0)
```

### **ClearListview(H)**

Zerstört alle Spalten und Zeilen eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls

Zurück bleibt ein ganz leeres Listview.

### **EditManual(H,S,Z,F)**

Erzeugt in einem Listview ein manuelles Editfeld, mit dem der User ein Listview-Item editieren kann.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
S : Long - Index der Spalte von H, in die das Editfeld gesetzt werden soll (nullbasierend).  
L : Long - Index der Zeile von H, in die das Editfeld gesetzt werden soll (nullbasierend).  
F : Long - Flags

Mit dieser Funktion ist es z.B. sehr einfach, beliebige Eingabemasken zu programmieren. Der User kann gezwungen werden, bestimmte Items in bestimmter Reihenfolge zu editieren.

H ist das Handle des Listviews, in dem editiert werden soll. S und L bestimmen das Item, in dem editiert werden soll.

Mit F können diverse Flags zugeschaltet werden. Momentan gibt es nur Flag 0:

Flag = 0: Normal editieren.

Flag = 1: Der Text im Editfeld ist bereits mit einer Selektierung markiert.

Es kann nur ein Edit gleichzeitig existieren (alles andere wäre auch nicht sinnvoll). Ob das Editfeld noch aktiv ist, kann mit [GetVar\(\)](#) und Flag 6 ermittelt werden.

Es empfiehlt sich, die User-gesteuerte Editier-Möglichkeit mittels [EnableEdits\(\)](#) auszuschalten, wenn manuelle Edits durchgeführt werden. Beide Arten arbeiten unabhängig voneinander.

```
EditManual(listview&,4,2,0)
```

### **GetVisibleColumns(H,B)**

Prüft, welche Spalten eines Listviews am linken und rechten Rand gerade sichtbar sind.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
B : Zeiger auf einen Bereich, der zwei Long-Ints aufnehmen kann.

Oft wünscht man sich zu wissen, welchen Spalten eines Listviews der User gerade betrachtet. Diese Informationen bietet das SysListview32 ansonsten nicht. GetVisibleColumns() erkennt, welche Spalten gerade im Listview (auch noch teilweise) sichtbar sind.

B muss ein Bereich sein, der zwei Long-Ints (acht Bytes) aufnehmen kann. In das erste Long-Int schreibt die Funktion den Index der ganz links sichtbaren Spalte, in das zweite Long-Int schreibt sie den Index der ganz rechts noch sichtbaren Spalte hinein.

Auf diese Weise erkennt der Programmierer genau, welchen Ausschnitt eines Listviews der User gerade betrachtet/bearbeitet.

Siehe auch [GetVisibleLines\(\)](#).

```
Dim bereich#,8  
GetVisibleColumns(listview&,bereich#)
```

```
x&=Long(bereich#,0)
y&=Long(bereich#,4)
Dispose bereich#
Print "Linke sichtbare Spalte: "+ Str$(x&)
Print "Rechte sichtbare Spalte: "+ Str$(y&)
```

### **GetVisibleLines(H,B)**

Prüft, welche Zeilen eines Listviews am oberen und unteren Rand gerade sichtbar sind.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
 B : Zeiger auf einen Bereich, der zwei Long-Ints aufnehmen kann.

Oft wünscht man sich zu wissen, welchen Zeilen eines Listviews der User gerade betrachtet. Diese Informationen bietet das SysListView32 ansonsten nicht. GetVisibleLines() erkennt, welche Zeilen gerade im Listview (auch noch teilweise) sichtbar sind.

B muss ein Bereich sein, der zwei Long-Ints (acht Bytes) aufnehmen kann. In das erste Long-Int schreibt die Funktion den Index der oberen sichtbaren Zeile, in das zweite Long-Int schreibt sie den Index der unteren noch sichtbaren Zeile hinein.

Auf diese Weise erkennt der Programmierer genau, welchen Ausschnitt eines Listviews der User gerade betrachtet/bearbeitet.

Siehe auch [GetVisibleColumns\(\)](#).

```
Dim bereich#,8
GetVisibleLines(listview&,bereich#)
x&=Long(bereich#,0)
y&=Long(bereich#,4)
Dispose bereich#
Print "Obere sichtbare Zeile: "+ Str$(x&)
Print "Untere sichtbare Zeile: "+ Str$(y&)
```

### **GetSelectedCount(H)**

Ermittelt, wieviele Zeilen momentan in einem Listview selektiert sind.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

Ergebnis: Long - Anzahl selektierter Zeilen.

### **DeleteColumn(H,S)**

Löscht eine komplette Spalte samt ihrem Inhalt.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
 S : Long - Index der Spalte von H, die gelöscht werden soll (nullbasierend).

Um alle Spalten zu löschen, also das ganze Listview zu leeren, wird [ClearListview\(\)](#) benutzt.

### **DeleteItem(H,L)**

Löscht eine Zeile samt ihrem Inhalt..

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
 L : Long - Index der Zeile von H, die gelöscht werden soll (nullbasierend).



### **DeleteAllItems(H)**

Löscht alle Zeilen eines Listviews.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

### **GetItemState(H,L,T)**

Ermittelt, ob eine Listview-Zeile einen bestimmten Status besitzt.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

L : Long - Index der Zeile von H, die untersucht werden soll (nullbasierend).

T : Long - Status, auf den die Zeile überprüft werden soll.

Ergebnis: Long - Null (0), wenn die Zeile diesen Status nicht besitzt. Grösser Null (>0), wenn der Status erfüllt ist.

Für T können folgende Werte eingesetzt werden:

1 = Die Zeile besitzt den Focus (kleines gepunktetes Rechteck umgibt um den Eintrag)

2 = Die Zeile ist ausgewählt (selektiert).

4 = Die Zeile ist für eine Ausschneiden/Einfügen Operation markiert.

8 = Die Zeile ist für eine Drag&Drop Operation markiert.

### **SetColumnWidth(H,S,P)**

Verändert die Breite einer Zeile.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

S : Long - Index der Spalte von H, die verändert werden soll (nullbasierend).

P : Long - Neue Breite der Spalte in Pixel.

Für P können beliebige positive Werte angegeben werden. Die neue Breite der Spalte beträgt nach dieser Funktion P Pixel..

Das System stellt aber noch zwei besondere - negative - Werte für P zur Verfügung, die folgende Bedeutung haben:

-1 = Die Spalte wird so breit gemacht, dass der längste Itemtext in die Spalte passt, ohne abgeschnitten zu werden.

-2 = Die Spalte wird so breit gemacht, dass die Spaltenüberschrift (Text des Spaltenbuttons) in die Spalte passt.

### **UpdateListview(H)**

Visuelle Erneuerung eines Listview.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.

Nach bestimmten Funktionen kann es nötig werden, dass das Listview neu gezeichnet werden muss, bevor die Anzeige aktualisiert wird. Das ist mit dieser Funktion jetzt einfach möglich, sollte eine Funktion mal nicht sofort reagieren.

### **HeaderToCsv(H,B,A,F)**

Auslesen der Spaltentexte eines Listviews in eine CSV-Datei.

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls  
B : Zeiger auf einen Bereich, in den alle Listview-Spaltenkopf-Texte als CSV-Datei gespeichert werden.  
A : Long - ASCII Code des Trennzeichens oder 0 (dann wird ein Komma verwendet)  
F : Long - Flag (0 = Texte in Anführungszeichen setzen / 1 = nur Texte).

Ergebnis: Long - Anzahl Bytes, die in den Bereich B geschrieben wurden.

Eine Funktion, um den Inhalt der Spaltenbuttons eines Listviews abzuspeichern. HeaderToCsv() schreibt den Inhalt des Headers des Listviews H als [CSV-Datei](#) in den Speicher B. Der Speicher muß natürlich groß genug dimensioniert sein, wie groß er mindestens sein muß, kann leicht berechnet werden mittels : Anzahl Spalten \* 264. Mit [WriteFileQuick\(\)](#) läßt er sich dann auf einem Datenträger abspeichern. In der CSV-Datei werden Kommas als Trennzeichen verwendet, mit A kann aber auch ein x-beliebiges Trennzeichen ausgewählt werden. Hierzu muß der ASCII-Wert des Zeichens angegeben werden. Hier mal ein paar alternative ASCII-Werte:

```
; = 59  
/ = 47  
- = 45
```

Aber Vorsicht, die Liestview.dll unterstützt nur Kommas oder Semikolons (wie fast alle anderen Programme auch, die CSV-Dateien benutzen).

Es ist aber jederzeit möglich, [ExchangeSeparator\(\)](#) zu verwenden, mit dem jedes beliebige Trennzeichen in einer CSV-Datei verarbeitet werden kann.

Mit F kann bestimmt werden, ob die einzelnen Itemtexte in Anführungszeichen gesetzt werden oder nicht. Das ist dann sinnvoll, wenn Spaltentexte Kommas oder Semikolons beinhalten, die ansonsten als Trennzeichen gewertet würden, wenn sie nicht in Anführungszeichen gesetzt sind. Das produziert dann natürlich böse Fehler...

HeaderToCsv(listview&,bereich#,0,0)

Siehe auch [ListviewToCsv\(\)](#), [CsvToHeader\(\)](#) und [CsvToListview\(\)](#).

### **SearchBlankItem(H,S,L)**

Sucht in einer Spalte nach freien Einträgen (leere Itemtexte).

H : Long - Handle eines mit [CreateListview\(\)](#) erstellten Listview Controls.  
S : Long - Index der Spalte von H, die gelöscht werden soll (nullbasierend).  
L : Long - Index der Zeile von H, ab der die Suche beginnen soll (nullbasierend).

Ergebnis: Long - Nullbasierende Zeile, in der ein freier Eintrag gefunden wurde oder -1, wenn keiner gefunden wurde.

Manchmal ist es nötig, freie Einträge - also leere Itemtexte - finden zu können. Dazu dient diese Funktion. Gesucht wird in Spalte S, wobei mittels L bestimmt werden kann, ab welcher Zeile die Suche beginnen soll. Sollen alle Spalten durchsucht werden, ist ein Durchlauf in einer Schleife nötig.

*Achtung, manche von XProfans DBase-Funktionen übergeben beim Auslesen anstelle von richtigen Leerstrings manchmal Strings, die mit Spaces gefüllt sind und nur dem Auge leer erscheinen. SearchBlankItem() findet diese nicht, da sie ja Zeichen enthalten. Ich könnte die Suche zwar dorthingehend erweitern, aber mitunter sind Effekte mit Hilfe von Leerzeichen ja erwünscht. Daher empfehle ich, Strings manuell zu trimmen, wenn diese mittels DBase-Funktionen ausgelesen wurden.*

Hier ein Beispiel, um alle leeren Itemtexte zu finden und mit einem Text zu füllen:

```

SendMessage(listview&,11,0,0) 'Neuaufbau des Listviews verhindern!

x&=0
y&=0
text$="Neuer Text"

While 1
  y&=SearchBlankItem(listview&,x&,y&)
  If y&=-1
    Inc x&
    Case (x&>=GetColumns(listview&)):BREAK
  Else
    SetItemText(listview&,addr(text$),x&,y&)
  Endif
  Inc y&
EndWhile

SendMessage(listview&,11,1,0) 'Wiederaufbau des Listviews (standart)

```

Siehe auch [SearchText\(\)](#).