Version 3.0

Autor:

Frank Abbing
Margarethenstr.13a
48683 Ahaus

Mail: frabbing@gmx.de
Webpage: http://frabbing.bplaced.net

Beschreibung

Eine DLL für alle Programmierer, die noch (viel) mehr aus ihrer Programmsprache herausholen wollen.
Die *ProSpeed.dll* ist sehr kompakt und in reinem Assembler programmiert.
Themen der DLL sind Sprites, Bildverarbeitung, Bild-Manipulationen, Png-mit-Alphakanal-Verarbeitung, Speicherbereiche, DirectSound, Pixeleffekte und allgemeine Spieleprogrammierung.

Die ProSpeed.dll wird unter Windows XP entwickelt und getestet, alle Funktionen arbeiteten bei den Tests einwandfrei. Sollten sie trotzdem noch einen Fehler finden, oder die Anleitung fehlerhaft sein, wäre es nett, wenn sie mir das mitteilten.

Alle Beispiele sind in Profan geschrieben und sollten ab Profan 7 funktionieren. Einige Beispiele zeigen auch Lösungsmöglichenkeiten für alte Profanversionen ab Version 5. Ich empfehle jedoch immer die aktuellste XProfan-Version.
Die Beispiele sind einfach gewählt und dürften schnell auch für andere Programmiersprachen portiert werden können, z.B. für Visual Basic usw.

Ab ProSpeed.dll Version 1.3 gibt es spezielle Sprite-Funktionen. Sie können Sprites generieren, die automatisch animiert und bewegt werden, ohne das sich das Hauptprogramm noch darum kümmern muß. Die Anzahl der Sprites ist unbegrenzt !

Das erinnert fast schon an Hardware-Sprites.
Eine spezielle Scrollfunktion sorgt (in Verbindung mit den Sprites) für eine Bewegung des Bildschirm-Hintergrunds (Scrolling).

Noch eine Sache zu den Sprites. Wenn sie diese benutzen, gibt es für das Hauptprogramm oft nichts anderes zu tun, als auf eine bestimmte Reaktion der Sprites innerhalb einer Schleife zu warten. In diesem Fall sollten sie auf jeden Fall den Sleep-Befehl oder die API Sleep() mit in die Schleife einbauen (z.B. Sleep 2), um der DLL mehr Rechenpower zuzuweisen. Das bescheunigt in vielen Fällen bis zu 100 Prozent.

Mit Version 3.0 erfolgte eine Neustrukturierung der Dll, in deren Zuge einige Funktionsnamen umbenannt wurden.

Ab Version 2.7 wurden mehrere Layer-Funktionen (um die Darstellungs-Ebene besser verändern zu können) dazu genommen, das war schon lange gefordert. Bleibt zu erwähnen, das beim ersten Sprite, welches erzeugt wird, der Layer nie geändert werden kann/darf.
Nur das zuerst generierte Sprite besitzt diesen Sonderstatus.

DirectSound ist ab ProSpeed 2.1 möglich. Im Gegensatz zu den originalen Abspiel-Funktionen für Wavefiles in Profan, können sie jetzt so viele Waves gleichzeitig abspielen, wie sie wollen. Mein besonderer Dank geht hier an Sebastian König (http://www.sekoenig.de), dessen DirectSound-Vorlagen mir eine große Hilfe waren. Damit ist auch in direktem Profan DirectSound möglich. Danke Sebastian !
Ab der Version 2.6 können Waves mit DirectSound auch aus dem Speicher oder aus einer Resource heraus geladen werden.

Ab Version 2.3 kann ProSpeed auch Bilder in den Formaten JPG, GIF und WMF laden. Ab Version 2.5 ist auch das PNG-Format dazu gekommen. Hinzu kommen die zwei Formate, die ja schon etwas länger ladbar sind, BMP und RLE.
Seit Version 3.0 gehen jetzt auch PNG-Dateien mit transparenten Masken. Dabei ist einer der großen Vorteile von PNG, dass die Bilder Transparenzinformationen enthalten können. Die ProSpeed.dll unterstützt hierbei alle Formate.
Ein Alphakanal ist eine zusätzliche Information, die für jedes Pixel angibt, wie viel vom Hintergrund des Bildes durchscheinen soll. ProSpeed unterstützt Alphakanäle von 8 Bit, was 256 Abstufungen der Transparenzstärke entspricht. Alphakanäle von 16 Bit werden intern auf 8 Bit herunter gerechnet. Somit ist es möglich, unabhängig vom Hintergrund die Kanten von Text und Bildern zu glätten. Man kann echte Schlagschatten verwenden, die im Hintergrund ausblenden, oder Bilder erzeugen, die beliebig geformt sind.

Mein Tip: Bilder mit Sprites sollten als BMP, PNG oder GIF gespeichert sein. JPG Bilder werden oft aufgepixelt abgespeichert und produzieren dadurch bei Sprites Transparenz-Probleme.
Hintergründe oder sonstige große Bilder sollten als JPG gespeichert sein, dieses Format erreicht die besten Komprimierungs-Raten und hält Programme schön klein. Auch PNG ist hierzu sehr nützlich.
Bilder können als Datei, direkt aus dem Speicher oder aus einer Resource-Datei (.exe oder .dll) geladen werden und ab ProSpeed 2.5 auch direkt im Quellcode stehen (Data / Oem).
Ab ProSpeed 2.7 können Bilder auch als Jpg gespeichert werden. Mit Version 3.0 kam die Möglichkeit, Png mit Alphakanal zu exportieren.

Ab Version 2.6 bietet die ProSpeed.dll die Funktionen Pack() und DePack() an. Die Technik, die hinter diesen Funktionen steht, stammt nicht aus meiner Feder. Es wird hier die aPLib von Joergen Ibsen verwendet, eine Library, die eine LZ komprimierende Technik verwendet, mehr dazu unten.
Wenn sie die ProSpeed.dll in Freewareprogramm benutzen, dürfen sie Pack() / Depack() / aPLib kostenlos benutzen, aber in Shareware Programmen und kommerizieller Software müssen sie aPLib registrieren.
Die Lizenz für aPLib erhalten sie für $ 29 (Firmen für $ 95) bei Joergen Ibsen (Webpage / Email).
Mit freundlicher Genehmigung. aPLib, copyright © 1998-2002 by Joergen Ibsen.

So, ich denke mit diesen mächtigen Funktionen sollte auch einem Anfänger ein schönes Spielchen gelingen. Wer noch Ideen hat, oder eine ganz wichtige Funktion vermißt, kann sich bei mir gerne melden. Ich bin auch bereit eine ganz individuelle Funktion aufzunehmen, wenn es mir sinnvoll erscheint und möglich ist. Meldet euch einfach bei mir...

Eigene grafische Effekte erstellen:

Hier die Beschreibung, wie man ein ByteArray - erzeugt mit InitFX() oder CreateFX() - selber manipulieren kann.
So ein Byte-Array ist nichts anderes als ein Bild, was aber komplett editierbar im Speicher liegt.
Man sollte aber ganz genau wissen, was man macht, weil ein versehentliches Schreiben in falsche Speicherbereiche beträchtlichen Schaden verursachen kann.
Darum habe ich mich bisher gescheut, diese Infos in die Anleitung aufzunehmen. Weil aber in letzter Zeit immer mehr Anwender die ProSpeed.dll nicht nur mit Profan nutzen, sondern auch selber Dll's schreiben oder schnellere Programmsprachen (z.B. PureBasic) verwenden und mit ProSpeed's grafischen Effekten nutzen möchten, habe ich dieses Kapitel hinzu gefügt.
Profan selber ist aber zu langsam, um eigene grafische Effekte vernünftig nutzen zu können.

InitFX() und CreateFX() übergeben einen Zeiger. Er zeigt auf eine originale Windows-Struktur (BITMAPINFOHEADER). Diese Struktur wurde allerdings von ProSpeed um ein Long-Int erweitert (Offset 40).
Hier die Offsets:

```
0  LONG biSize
4  LONG biWidth, Breite des Bildes
8  LONG biHeight, Höhe des Bildes
12 WORD biPlanes, immer 1
14 WORD biBitCount, bei ProSpeed Byte-Arrays immer 24 (Bit)
16 LONG biCompression, immer BI_RGB
20 LONG biSizeImage, Größe des Byte-Arrays in Bytes
24 LONG biXPelsPerMeter
28 LONG biYPelsPerMeter
32 LONG biClrUsed
36 LONG biClrImportant
40 LONG Zeiger auf den Speicher des Byte-Arrays
```

Auf die Werte der Struktur kann von Profan (ab 7) aus mit @Long zugegriffen werden..
Am interessantesten ist natürlich das Long-Int an Offset 40. Hier steht ein Zeiger auf eigentlichen Grafikdaten.

Bei früheren Versionen der ProSpeed.dll war die Grafik noch auf den Kopf gestellt. Ab Version 3.0 nicht mehr, und die Bitmap in einem Byte-Array wird von links oben nach rechts unten abgespeichert.
Pro Pixel werden 3 Byte benötigt (24-Bit), ein Blau-Wert, ein Grün-Wert und ein Rot-Wert (in dieser Reihenfolge BGR, jeweils ein Byte). Wenn die Breite der Grafik im Bytearray nicht glatt durch vier teilbar ist, fügt Windows zusätzliche Füllbytes ins Bytearray ein, was das Manipulieren der Grafikdaten ungleich erschwert und durch zusätzliche Abfragen verlangsamt. Darum verlangen alle grafischen Effekte der ProSpeed.dll ein Bild, dessen Breite glatt durch vier teilbar ist!

Nachdem ihr das ByteArray manipuliert habt, könnt ihr es mit CopyFX() auf den Bildschirm bringen.

Infos für andere Programmiersprachen

Wenn irgenwo 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.

Funktionen der ProSpeed.dll
(Logisch geordnet nach Themen.)

Externe Bitmaps:

CreateImage
LoadFileImage
LoadMemoryImage
LoadResourceImage
ColorFillImage
Curve
Frame
FreeAllImages
FreeImage
GetHeightImage
GetInfosImage
GetWidthImage
PaintImage
RotateImage
SaveImage
SizeImage

Grafische Effekte mit externen Bitmaps:

CreateFX
InitFX
AlphaTrans *erweitert!*
Antialiasing
BitBltArray *neu!*
BlackWhite
Blur
CloneFX
ColorFill *neu!*
ColorMove
CopyFX
Darken
ExchangeColor
ExchangeRgb *neu!*
FlipX
FlipY
Fog
FreeFX
Grey
Lighten
Merge
Rotate
Rotate180
Rustle
SaveFX
SaveToMemoryFX
SemiTrans
Sharpen
Smooth
Water *erneuert!*

Sound (benutzt DirectX):

Funktionen der ProSpeed.dll
(In alphabethischer Reihenfolge)

AddWindowStyle
AlphaTrans
Antialiasing
AttachSprite
AttachSpriteAnim
BitBltArray
BlackWhite
Blur
BringSpriteToBottom
BringSpriteToTop
ChangeBytesToWords
ChangeDisplay
ChangeLongSequence
ChangeSpritePara
ChangeWordsToLongs
CleanUp
ClearWith
CloneFX
Collide
CollideAll
CollideMore
CollideUnknown
Collision
ColorFill
ColorFillImage
ColorMove
CompareBytes
Compute
CopyFX
CopyMemory
CopyMemoryFromBack
CopySprite
CountAllSprites
CountBytes
CountFirstBytes
CountJoystickButtons
CountLongs
CountSprites
CountStrings
CountWords
CreateFX
CreateImage
Crypt
Curve
Darken
DataBytes (Data)
DataMem
DataPos
DeInitPixelEffects
DeleteAllSprites
DeleteAllXSprites
DeleteAllYSprites
DeleteSprite
DeleteSpritesIfAnimReady
DeleteSpritesMark
DeleteTags

DSoundInit
DSoundDeInit
DSoundGetInterfaces
DSoundGetNextBuffer
DSoundGetStatus
DSoundLoad
DSoundLoadMemory
DSoundLoadResource
DSoundPlay
DSoundSetGlobalPan
DSoundSetGlobalVolume
DSoundSetPan
DSoundSetVolume
DSoundStop
DSoundUnLoad

Komprimierung:

Pack
DePack
GetOriginalSize

Sprites:

InitSprite
InitSprite5
AttachSprite
AttachSpriteAnim
BringSpriteToBottom
BringSpriteToTop
ChangeSpritePara
CleanUp
Collide
CollideAll
CollideMore
CollideUnknown
Collision
CopySprite
CountAllSprites
CountSprites
DeleteAllSprites
DeleteAllXSprites
DeleteAllYSprites
DeleteSprite
DeleteSpritesIfAnimReady
DeleteSpritesMark
ExistSprite
GetBackgroundInfos
GetSpriteAnimModeStatus
GetSpriteFixMode
GetSpriteInfos
GetSpriteLayer
GetSpriteMark
GetSpriteMoveX
GetSpriteMoveXY
GetSpriteMoveY
GetSpritesFps
GetSpriteX

DePack
DownloadHttpFile
DSoundDeInit
DSoundGetInterfaces
DSoundGetNextBuffer
DSoundGetStatus
DSoundInit
DSoundLoad
DSoundLoadMemory
DSoundLoadResource
DSoundPlay
DSoundSetGlobalPan
DSoundSetGlobalVolume
DSoundSetPan
DSoundSetVolume
DSoundStop
DSoundUnLoad
Examine
ExchangeColor
ExchangeRgb
ExistSprite
ExportPng
FindBytes
FindBytesCount
FindJoystick
FindPath
FlipX
FlipY
Fog
Frame
FreeAllImages
FreeFX
FreeImage
FreePng
FXToPng
GenerateChecksum
GetBackgroundInfos
GetHeightImage
GetInfosImage
GetOriginalSize
GetSpriteAnimModeStatus
GetSpriteFixMode
GetSpriteInfos
GetSpriteLayer
GetSpriteMark
GetSpriteMoveX
GetSpriteMoveXY
GetSpriteMoveY
GetSpritesFps
GetSpriteX
GetSpriteY
GetSystemKey
GetVersion
GetWidthImage
Grey
HasSpriteArrived
ImportPng
InitFX
InitPixelEffects
InitSprite

JoystickX
JoystickY
JoystickZ
MouseButton
PMouse
SetMouseRect
SetMouseXY

Bereiche / Tabellen:

ChangeBytesToWords
ChangeLongSequence
ChangeWordsToLongs
ClearWith
CompareBytes
Compute
CopyMemory
CopyMemoryFromBack
CountBytes
CountFirstBytes
CountLongs
CountStrings
CountWords
Crypt
DataBytes (Data)
DataMem
DataPos
DeleteTags
Examine
FindBytes
FindBytesCount
GenerateChecksum
LongToAddrVar4
OemBytes (Oem)
PixelToMemory
Random
ReadFileFast
ReadFilePieceFast
Replace
ReplaceTabs
RotateMem
SetBytes
SetLongs
SetWords
VarToLong10
VarToLong4
WriteFileFast

Timer:

StartWatch
ReadWatch
WaitWatch

Sonstiges:

AddWindowStyle
SetSpriteMovingMode
SetSpritePos
SetSpriteSpeed
SetTiles
SetTiles5
SetWords
Sharpen
SizeImage
SlowDownSprite
Smooth
SmoothPath
SpriteMovingRect
SpriteScrollMode
SpriteTableMode
SpriteThreadBrake
SpriteToHDC
Stars
StartWatch
StatusStars
StopAllSprites
SubWindowStyle
SwapSpriteLayers
SwitchTaskbar
Tooltip
VarToLong10
VarToLong4
Version
VisualTableFix
WaitWatch
Water
WriteFileFast

Funktionen der ProSpeed.dll
(Kompakt ABC)

AddWindowStyle AlphaTrans Antialiasing AttachSprite AttachSpriteAnim BitBltArray BlackWhite Blur BringSpriteToBottom BringSpriteToTop ChangeBytesToWords ChangeDisplay ChangeLongSequence ChangeSpritePara ChangeWordsToLongs CleanUp ClearWith CloneFX Collide CollideAll CollideMore CollideUnknown Collision ColorMove CompareBytes Compute CopyFX CopyMemory CopyMemoryFromBack CopySprite CountAllSprites CountBytes CountFirstBytes CountJoystickButtons CountLongs CountSprites CountStrings CountWords CreateImage CreateFX Crypt Curve Darken DataBytes (Data) DataMem DataPos DeInitPixelEffects DeleteAllSprites DeleteAllXSprites DeleteAllYSprites DeleteSprite DeleteSpritesIfAnimReady DeleteSpritesMark DeleteTags DePack DownloadHttpFile DSoundDeInit DSoundGetInterfaces DSoundGetNextBuffer DSoundGetStatus DSoundInit DSoundLoad DSoundLoadMemory DSoundLoadResource DSoundPlay SoundSetGlobalPan DSoundSetGlobalVolume DSoundSetPan DSoundSetVolume DSoundStop DSoundUnLoad Examine ExchangeColor ExchangeRgb ExistSprite ExportPng ColorFillImage ColorFill FindBytes FindBytesCount FindJoystick FindPath FlipX FlipY Fog Frame FreeAllImages FreeImage FreeFX FreePng FXToPng GenerateChecksum GetBackgroundInfos GetHeightImage GetInfosImage GetWidthImage GetOriginalSize GetSpriteAnimModeStatus

ChangeDisplay
DownloadHttpFile *neu!*
GetSystemKey
GetVersion
IsInternet *neu!*
SubWindowStyle
SwitchTaskbar
Tooltip
Version

Png:

ExportPng *neu!*
FreePng *neu!*
FXToPng *neu!*
ImportPng *neu!*
InvertChannelPng *neu!*
MergeChannelPng *neu!*
MergeImagePng *neu!*
PaintPng *neu!*
PngToFX *neu!*

GetSpriteFixMode GetSpriteInfos GetSpriteLayer GetSpriteMark GetSpriteMoveX GetSpriteMoveXY GetSpriteMoveY GetSpritesFps GetSpriteX GetSpriteY GetSystemKey GetVersion Grey HasSpriteArrived ImportPng InitFX InitPixelEffects InitSprite InitSprite5 InitSpriteBackground InvertChannelPng IsInternet JoystickButton JoystickR JoystickU JoystickV JoystickX JoystickY JoystickZ Lighten LoadFileImage LoadMemoryImage LoadResourceImage LongToAddrVar4 MarkSprite Merge MergeChannelPng MergeImagePng MouseButton MouseOverSprite MoveSprite MoveSpriteWithTable NoStars OemBytes (Oem) OptionStars Pack PaintImage PaintPng PixelToMemory PMouse PngToFX Random ReadFileFast ReadFilePieceFast ReadWatch Replace ReplaceTabs Rotate Rotate180 RotateImage RotateMem Rustle SaveFX SaveImage SaveToMemoryFX SemiTrans SetBackAutoCollision SetBytes SetCollideUnknownSprite SetLongs SetMouseRect SetMouseXY SetPixelEffect SetSpriteAnim SetSpriteAnimMode SetSpriteAnimMove SetSpriteCage SetSpriteFixMode SetSpriteLayer SetSpriteMovingMode SetSpritePos SetSpriteSpeed SetTiles SetTiles5 SetWords Sharpen SizeImage SlowDownSprite Smooth SmoothPath SpriteMovingRect SpriteScrollMode SpriteTableMode SpriteThreadBrake SpriteToHDC Stars StartWatch StatusStars StopAllSprites SubWindowStyle SwapSpriteLayers SwitchTaskbar Tooltip VarToLong10 VarToLong4 Version VisualTableFix WaitWatch Water WriteFileFast

IsInternet()

Prüft, ob eine Internetverbindung aufgebaut ist. Siehe auch DownloadHttpFile().

Ergebnis: Long - Wenn eine Internetverbindung geöffnet ist 1, bei fehlender Verbindung 0.

DownloadHttpFile(U,D)

Lädt eine Datei aus dem Internet auf Festplatte. Siehe auch IsInternet().

```
U : String (oder Bereich), der den Namen der Http-Datei enthält
D : String (oder Bereich), der den Dateinamen der zu schreibenden Datei enthält
```

Ergebnis: Long - 0, bei Fehler <> 0.

```
IfNot DownloadHttpFile("http://frabbing.bplaced.net/bg.gif", "c:/test.txt")
  Print "Datei wurde generiert."
Else
  Print "Download gescheitert!"
EndIf
```

ColorFill(A,F)

Färbt ein Effekt-Byte-Array in einer bestimmten Farbe.

```
A : Long - Ziel-Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays
F : Long - RGB-Farbwert
```

Eine sehr simple Funktion um ein ByteArray schnell einzufärben.

```
ColorFill(array&,Rgb(126,34,220))
```

ExchangeRgb(F,X,Y,A,R,G,B,)

Verstärkt sie, oder schwächt die Farbkanäle Rot, Grün und/oder Blau ab. Also eine Art Lighten(), bzw. Darken(), aber auch einsetzbar für einzelne Farbkanäle.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
A : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
R : Long - Verstärkung (plus) oder Abschwächung (minus) für roten Farbkanal
(Werte von -255 bis 255)
G : Long - Verstärkung (plus) oder Abschwächung (minus) für grünen Farbkanal
(Werte von -255 bis 255)
B : Long - Verstärkung (plus) oder Abschwächung (minus) für blauen Farbkanal
(Werte von -255 bis 255)
```

Ergebniss: Long - 0.

Die einzelnen Farbkomponenten Rot, Grün und Blau können mit dieser Funktion verstärkt oder abgeschwächt werden.

X und Y geben die linke obere Ecke an, ab der der Effekt auf dem Bildschirm gebracht wird. Die Grösse des Effekts ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
R, G und B nehmen Werte von -255 bis 255 an. Wobei Minuswerte bewirken, dass dieser Kanal abgeschächt wird, und Pluswerte, dass der Kanl verstärkt wird. Eine Null bedeutet, dieser Kanal wird nicht verändert. Je weiter die Werte R, G und B sich von Null entfernen, desto sichtbarer wird der Effekt.
Bevor ExchangeRgb() angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben. Die Daten im Byte-Array A werden durch Aufruf dieser Funktion verändert.
ExchangeRgb() ersetzt im Grunde auch Darken() und Lighten(), weil diese Effekte mit der Funktion sehr einfach simuliert werden können.

```
ExchangeRGB(0,0,0,array&,1,0,0)        'Rot leicht verstärken
ExchangeRGB(0,0,0,array&,0,0,-20)      'Blau stark abschwächen
ExchangeRGB(0,0,0,array&,255,255,255) 'Komplett weiss färben
```

ImportPng(D,B,H,A)

Importiert PNG-Dateien mit 8/16-Bit-Alphakanal.

```
D : String (oder Bereich), der den Dateinamen der PNG-Datei enthält
B : Long - Zeiger auf ein Long, welches die Breite des geladenen Bilds aufnimmt
H : Long - Zeiger auf ein Long, welches die Höhe des geladenen Bilds aufnimmt
A : Long - Zeiger auf ein Long, dass die Größe des Bilds in Bytes aufnimmt
```

Ergebnis: Long - Png-Handle, bei Fehler 0.

Seit Version 2.5 kann die ProSpeed.dll bereits Png-Dateien laden, Auch XProfan kann das seit Version 11. Was aber bislang nicht ging, waren Png's mit transparenten Masken einzuladen und anzuzeigen. Dabei ist einer der großen Vorteile von PNG-Dateien, dass die Bilder Transparenzinformationen enthalten können. Die ProSpeed.dll unterstützt hierbei nun den weit verbreiteten Alphakanal 8.
Ein Alphakanal ist eine zusätzliche Information, die für jedes Pixel angibt, wie viel vom Hintergrund des Bildes durchscheinen soll. ProSpeed unterstützt Alphakanäle von 8 und 16 Bit, was 256 Abstufungen der Transparenzstärke entspricht (16 Bit wird intern auf 8 Bit herunter herunter gerechnet). Somit ist es jetzt möglich, unabhängig vom Hintergrund die Kanten von Text und Bildern zu glätten. Man kann echte Schlagschatten verwenden, die im Hintergrund ausblenden, oder Bilder erzeugen, die beliebig geformt sind...
Png's mit Maske können nicht wie andere Bilder geladen werden. Hierzu ist diese Funktion nötig. ImportPng() benötigt dafür vier Parameter:
D ist ein Zeiger auf einen Ordner- und Dateinamen des Png-Bilds.
B und H sind Adressen von Variablen, welche Breite und Höhe des importierten Bilds speichern.
A ist die Adresse einer Variablen, welche die Anzahl Bytes speichert, die das Bild belegt.

Die Funktion liefert den Handle des importierten Png-Bilds zurück, der für die anderen Png-Funktionen benötigt wird. Dieser Handle ist aber nicht kompatibel zu Handles von Bildern, die z.B. mit LoadFileImage() ermittelt wurden! Der Handle muß spätestens zum Programmende freiegegeben werden mittels FreePng().

Achtung! Wie bei den grafischen Effekten sollten auch die Png-Bilder Breiten besitzen, die glatt durch 4 teilbar sind!

Beispiel:

```
text$="name.png"
pnghandle&=ImportPng(addr(text$),addr(breite&),addr(hoehe&), addr(anzahlbytes&))
Print breite&, hoehe&, anzahlbytes&
```

ExportPng(P,D,B,H)

Exportiert PNG-Dateien mit 8/16-Bit-Alphakanal.

```
P : Long - Zeiger auf ein Png-Handle, erstellt mit ImportPng() oder PngToFX()
D : String (oder Bereich), der den Dateinamen der PNG-Datei enthält
B : Long - Breite des Png-Bilds P
H : Long - Höhe des Png-Bilds P
```

Ergebnis: Long - Anzahl der geschriebenen Bytes in Datei D.

Diese Funktion exportiert eine Png-Datei aus dem Png-Handle, der in P übergeben wurde. D ist ein Zeiger auf einen Ordner- und Dateinamen des zu schreibenden Png-Bilds.
Da der Png-Handle nur die eigentlichen Daten des Png-Bilds enthält, sind zusätzliche Angaben zum Exportieren nötig: B und H sind die Dimensionen des Bildes.

Beispiel:

```
text$="name.png"
pnghandle&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

text$="neuername.png"
anzahlbytes&=ExportPng(pnghandle&,addr(text$),breite&,hoehe&)
```

FreePng(P)

Gibt die Resourcen eines Pngs wieder frei.

```
P : Long - Zeiger auf ein Png-Handle, erstellt mit ImportPng() oder PngToFX()
```

PaintPng(P,A,Z,X,Y,B,H,Q)

Kopiert ein Png-Bild mit Alphakanal auf einen HDC.

```
P : Long - Zeiger auf ein Png-Handle, erstellt mit ImportPng() oder PngToFX()
A : Long - Größe des Bilds P in Bytes, wurde von ImportPng() in A übergeben
H : Long - Handle der Quell- und Zielbitmap (HDC)
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Breite des Png-Bilds P
H : Long - Höhe des Png-Bilds P
Q : Long - 0 oder Handle auf eine Quellbitmap (HDC)
```

Ergebnis: Long - TRUE, bei Fehler 0.

Diese Funktion kopiert ein Png mit Alphakanal auf einen HDC.
PaintPng() benötigt für seine Arbeit zwei Quellen, nämlich P und H. Wenn Q aber ungleich 0 ist, dann werden P und Q als Quellen verwendet. Q kann also als DoubleBuffering benutzt werden, damit das Ergebnis dieser Funktion die Quelle für weitere Aufrufe nicht überschreibt (ist Q gleich Null, dann bleibt dieser Parameter unbenutzt).
Eine Quelle ist also P, die anderen H (bzw. Q). PaintPng() kombiniert jetzt beide anhand des Alphakanals (in P) und generiert aus den zwei Bildern eines, und zeichet es ins Ziel H an Pixel-Position X und Y.

Die Breite B und Höhe H müssen die Dimension des Png-Bilds beinhalten.
A ist die Größe des Pngs in Bytes. Der Wert, den ImportPng() unter A zurück liefert.

Wichtig: Die Breite von Bild P muß ohne Rest durch vier teilbar sein!

Beispiel:

```
text$="png.png"
handle&=ImportPng(addr(text$),addr(x&),addr(y&),addr(bytes&))

If handle&

 WhileLoop 156
  PaintPng(handle&,bytes&,%hdc,-512+(&loop*8),20,x&,y&,%hdc2)
 EndWhile

 FreePng(handle&)

Endif
```

InvertChannelPng(P,A)

Invertiert den Alphakanal eines Png-Bilds.

```
P : Long - Zeiger auf ein Png-Handle, erstellt mit ImportPng() oder PngToFX()
A : Long - Größe des Bilds P in Bytes, wurde von ImportPng() in A übergeben
```

Die Transparenz-Werte eines 8-Bit-Alphakanals reichen von 0 bis 255. Diese Werte werden durch die Funktion einfach invertiert.

FxToPng(Q,B,H,A)

Erstellt aus einer (bzw. zwei) Bytes-Array Struktur(en) (grafischer Effekt) ein Png-Handle (mit 8-Bit Alphakanal).

```
Q : Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-Arrays
B : Long - Zeiger auf ein Long, welches die Breite des neuen Png-Bilds aufnimmt
H : Long - Zeiger auf ein Long, welches die Höhe des neuen Png-Bilds aufnimmt
A : Long - Zeiger auf ein Long, dass die Größe des neuen Png-Bilds in Bytes
aufnimmt
M : 0 oder Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays
```

Ergebnis: Long - Png-Handle, bei Fehler 0.

Ab Version 3.0 unterstützt die ProSpeed.dll Png-Dateien mit dem weit verbreiteten 8-Bit Alphakanal. Um diese Dateien kompatibel zu machen mit den grafischen Effekten, gibt es diese Funktion.

Q ist der Handle eines ByteArrays. B und H sind Adressen von Variablen, welche Breite und Höhe des neuen Png-Bilds speichern. A ist die Adresse einer Variablen, welche die Anzahl Bytes speichert, die das neue Bild im Speicher belegt.

Die Funktion liefert den Handle des neu entstandenen Png-Bilds zurück, der für die anderen Png-Funktionen benötigt wird. Der Alphakanal wird für alle Pixel gefüllt, sodass das ganze Bild bei der Anzeige mit

PaintPng() komplett sichtbar ist. Ist M ungleich 0, dann wird das Bild in M als Alphakanal-Maske verwendet. Q und M müssen in dem Fall gleiche Dimensionen besitzen.
Der zurück gegebene Handle ist nicht kompatibel zu Handles von Bildern, die z.B. mit LoadFileImage() ermittelt wurden! Und er muß spätestens zum Programmende freiegegeben werden mittels FreePng().

Beispiel:

```
text$="background.bmp"
hdc&=LoadFileImage(Addr(text$))

text$="maske.bmp"
hdc2&=LoadFileImage(Addr(text$))

array&=InitFX(hdc&)
array2&=InitFX(hdc2&)

pnghandle&=FXToPng(array&,addr(breite&),addr(hoehe&),addr(anzahlbytes&),array2&)

If pnghandle&
PaintPng(pnghandle&,anzahlbytes&,%hdc,0,0,breite&,hoehe&,0)
FreePng(pnghandle&)
Endif

FreeFX(array&)
FreeFX(array2&)
FreeImage(hdc&)
FreeImage(hdc2&)
```

PngToFX(P,B,H,A,M)

Erstellt aus einem Png-Handle mit Alphakanal ein (bzw. zwei) Bytes-Array Struktur(en) für grafischer Effekte.

```
P : Long - Zeiger auf ein Png-Handle, erstellt mit ImportPng() oder PngToFX()
B : Long - Breite des Png-Bilds P
H : Long - Höhe des Png-Bilds P
A : Long - Größe des Bilds P in Bytes, wurde von ImportPng() in A übergeben
M : Long - 0 oder Zeiger auf ein Long, welches den Handle einer weiteren Bytes-
Arrays Struktur zurück liefert
```

Ergebnis: Long - Handle einer Byte-Array Struktur, bei Fehler 0.

Ab Version 3.0 unterstützt die ProSpeed.dll Png-Dateien mit dem weit verbreiteten 8/16-Bit Alphakanal. Um diese Dateien kompatibel zu machen mit den grafischen Effekten, gibt es diese Funktion.

P ist der Handle eines Png's mit Alphakanal. B und H geben dessen Breite an. A ist die Größe des Png's in Bytes, also der Wert, den ImportPng() unter A zurück liefert.
Das Ergebnis dieser Funktion ist ein ByteArray eines grafisches Effekts mit den reinen Bilddaten aus P und den Ausmassen B mal H.
Ist M ungleich 0, dann wird in M ein weiteres Byte-Array zurück gegeben mit den Alphakanal-Daten aus P. Dieser Byte-Array Handle und der aus dem Ergebniss der Funktion müssen spätestens zum Programmende mit FreeFX() freigegeben werden.

Beispiel:

```
text$="png.png"
```

```
pnghandle&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

If pnghandle&

 array&=PngToFX(pnghandle&,breite&,hoehe&,anzahlbytes&,addr(maskarray&))
 FreePng(pnghandle&)

 If array&
  CopyFX(%HDC,0,0,array&)
  CopyFX(%HDC,%maxx-breite&,%maxy-hoehe&,maskarray&)

  FreeFX(array&)
  FreeFX(maskarray&)

 Endif

Endif
```

MergeChannelPng(P,Z,A,S)

Nähert den Alphakanal aus einem Png-Handle dem Alphakanal eines anderen Png-Handles an. Zusammenführung für Grafikmasken also.

P : Long - Zeiger auf ein Png-Handle (Quelle), erstellt mit ImportPng() oder PngToFX()
Z : Long - Zeiger auf ein Png-Handle (Ziel), erstellt mit ImportPng() oder PngToFX()
A : Long - Größe der Bilder P und Z in Bytes, wurde von ImportPng() in A übergeben
S : Long - Fortschritt der Annäherung (1-32)

Ergebnis: Long - Anzahl der Annäherungen. Bei 0 sind beide Alphakanäle identisch.

Hiermit sind nette Effekte möglich. Die Grafik in P und Z wird nicht geändert, nur der Alphakanal von Z wird dem Alphakanal von P ähnlicher gemacht.
S gibt an, ob diese Annäherung schnell vonstatten gehen soll oder nicht, und bedeutet eine Anzahl Schritte. Nach spätestens 256 Schritten wurde die Annäherung komplett vollzogen. Das Ende der Annäherung sollte aber immer mittels des Ergebnis dieser Funktion geprüft werden.
Siehe auch MergeImagePng().

Beispiel:

```
text$="png3.png"
pnghandle1&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

text$="png4.png"
pnghandle2&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

If pnghandle1&

 WhileLoop 256

  PaintPng(pnghandle1&,anzahlbytes&,%hdc,&loop*3,&loop,breite&,hoehe&,%hdc2)
  Inc x&
```

```
 z&=MergeChannelPng(pnghandle1&,pnghandle2&,anzahlbytes&,2)
 Case z&=0: BREAK

EndWhile

FreePng(pnghandle1&)
FreePng(pnghandle2&)

EndIf
```

MergeImagePng(P,Z,A,S)

Nähert die Bilddaten aus einem Png-Handle den Bilddaten eines anderen Png-Handles an.

P : Long - Zeiger auf ein Png-Handle (Quelle), erstellt mit ImportPng() oder PngToFX()
Z : Long - Zeiger auf ein Png-Handle (Ziel), erstellt mit ImportPng() oder PngToFX()
A : Long - Größe der Bilder P und Z in Bytes, wurde von ImportPng() in A übergeben
S : Long - Fortschritt der Annäherung (1-32)

Ergebnis: Long - Anzahl der Annäherungen. Bei 0 sind beide Alphakanäle identisch.

Hiermit sind nette Effekte möglich. Die Grafik in P wird der Grafik in P ähnlicher gemacht.
S gibt an, ob diese Annäherung schnell vonstatten gehen soll oder nicht, und bedeutet eine Anzahl Schritte. Nach spätestens 256 Schritten wurde die Annäherung komplett vollzogen. Das Ende der Annäherung sollte aber immer mittels des Ergebnis dieser Funktion geprüft werden.
Siehe auch MergeChannelPng().

Beispiel:

```
text$="png3.png"
pnghandle1&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

text$="png4.png"
pnghandle2&=ImportPng(addr(text$),addr(breite&),addr(hoehe&),addr(anzahlbytes&))

If pnghandle1&

 WhileLoop 256

  PaintPng(pnghandle1&,anzahlbytes&,%hdc,&loop*3,&loop,breite&,hoehe&,%hdc2)
  Inc x&

  z&=MergeImagePng(pnghandle1&,pnghandle2&,anzahlbytes&,2)
  Case z&=0: BREAK

 EndWhile

 FreePng(pnghandle1&)
 FreePng(pnghandle2&)
```

```
EndIf
```

BitBltArray(Z,X1,Y1,B,H,Q,X2,Y2,F,V)

Kopiert einen rechteckigen Ausschnitt von einem Byte-Array in ein anderes. Ggf mit mit transparenter Farbe oder Maske.

```
Z : Long - Ziel-Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays
X1: Long - linke Koordinate im Zielarray
Y1: Long - obere Koordinate im Zielarray
B : Long - Breite des Rechtecks, das kopiert werden soll
H : Long - Höhe des Rechtecks, das kopiert werden soll
Q : Long - Quell-Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays
X2: Long - linke Koordinate im Quellarray
Y2: Long - obere Koordinate im Quellarray
F : Long - Flag
V : Long - zusätzliche Variable für bestimmte Flag F.
```

Der rechteckige Ausschnitt des Quellarrays beginnend bei X2,Y2 der eine Breite von B Pixel und eine Höhe von H Pixel hat, wird an die Position X1,Y1 in das Zielarray kopiert.
Der letzte Parameter F bestimmt, in welchem Modus kopiert werden soll. Ist F=0, dann wird der rechteckige Bereich ganz normal kopiert.
Ist F=1, dann muss in V eine transparente Farbe angegeben werden, z.B. RGB(255,255,255). Besitzt ein Pixel in der Quellgrafik diese Farbe, wird er nicht ins Ziel kopiert.
Ein weiteres Flag hat den Wert 2. Ist F=2, dann muss in V ein weiterer Handle eines Byte-Arrays angegeben werden. V ist eine Maskengrafik und bestimmt, ob die Pixelfarbe von Z oder Q dominieren soll. Ist der Pixelwert in V=0 (schwarz), dann wird die Pixelfarbe der Quelle verwendet. Ist V=255 (weiß), dann wird die Pixelfarbe des Ziels verwendet. Ist V=127, dann werden Quell- und Zielfarbe zu gleichen Teilen vermischt verwendet. Je schwärzer ein Pixel in der Maske also ist, desto dominanter die Quellfarbe, je weisser, desto dominanter die Zielfarbe
Wichtig! Ist F=2, dann muß die Maskengrafik die Größe B x H besitzen!
Diverses Clipping (Grafiken gehen über Ränder hinaus) muß von User selber verhindert werden. Die Funktion prüft das nicht nach.
Siehe auch AlphaTrans().

FindBytesCount (B,O,A,S,L,C)

Eine Erweiterung der Funktion FindBytes(). Sucht aber nicht nach den ersten Vorkommen eines Strings, sondern einem beliebigen.

```
B : Long - Bereichsvariable
O : Long - Offset zum Bereich
A : Long - Anzahl zu durchsuchender Bytes im Bereich
S : Long - Zeiger auf einen Suchstring (ab Profan 7) oder zweite
Bereichsvariable, z.B. Addr(string$)
L : Long - Länge des Strings oder zweiten Bereichs, z.B. Len(string$)
C : Long - Nummer des Vorkommens von S. Nullbasierend
```

Ergebniss: Long - Fundstelle oder -1, wenn der String nicht gefunden wird.
Mit dieser Funktion kann eine beliebige Zeichenfolge in einem Bereich gesucht werden. Hierbei kann angegeben werden, die wievielte Fundstelle gemeint ist.

Beispiel, um eine bestimmte Zeile in einer Textdatei zu finden:

```
text$="Dateiname"           'Dateiname
bytes&=FileSize(text$)      'Länge der Datei ermitteln (hier als Beispiel 48030
Bytes)

Dim bereich#,bytes&         'Bereich für Datei dimensionieren

ReadFileFast(addr(text$),bereich#,0,bytes&)             'Datei laden

text$=Chr$(13)                                          'Suchstring (CR).
x&=FindBytesCount(bereich#,0,bytes&,Addr(text$),1,146)  'Zeile 146 suchen
x&=x&+2                                                 'CR und LF überspringen
```

Merge (Q,Z,F)

Verschmelzung/Kombinierung von zwei Bildern.

```
Q : Long - Quell-Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays
Z : Long - Ziel-Handle eines mit InitFX() oder CreateFX() erzeugten Bytes-Arrays
F : Long - Flags
```

Ergebniss: Long - 0.

Die Grafik des Byte-Arrays Q wird in die Grafik von Z kopiert. Q und Z müssen gleiche Ausmessungen besitzen. Nach Z werden nur die Pixel von Q kopiert, welche nicht die transparente Farbe besitzen. Wobei der obere, rechte Pixel von Q den Farbwert der transparenten Farbe bestimmt.
F beinhaltet die Flags. Ist F = 0, dann wird das Bild normal kopiert. Ist F = 1, wann werden zusätzlich noch die Kanten der Grafik geglättet, die kopiert wird (Antialiasing). Ist F = 3, dann wird die Kante der Grafik nicht mitkopiert. Mit den Flags 1 und 3 werden Grafiken fast nahtlos mit der Zielgrafik verschmolzen.
Das verschmolzene Bild befindet sich nach Aufruf von Merge() in Z. Um es auf den Bildschirm zu bringen, kann CopyFX() verwendet. Die Grafik in Q wird nicht verändert.

CloneFX (H)

Erzeugt eine Kopie eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-Arrays. Die Struktur wird kopiert, ebenso wie die eigentliche Grafik im Array.

```
H : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays
```

Ergebniss: Long - Handle des erzeugten Byte-Arrays.

Das Handle des kopierten Arrays muss später, wie gewohnt, mit FreeFX() freigegeben werden!

Antialiasing (H,X,Y,B,R,F,A)

Verwischt die Ränder einer Grafik oder wahlweise die ganze Grafik und versucht den Treppeneffekt dadurch einzudämmen..
Deutlich langsamer als z.B. Blur(), wenn nur die Ränder vom Effekt betroffen sind.

```
H : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
```

```
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays
R : Long - RGB-Farbwert der transparenten Farbe oder -1
F : Long - Flags
A : Long - Anzahl der Durchläufe des Effekts
```

Ergebniss: Long - 0.

Die eigentliche Grafik und der transparente Teil des Bildes B werden soft miteinander vermischt (Antialiasing). Somit werden harte Kanten geglättet.
X und Y geben die linke obere Ecke an, an die das Bild B später in H dargestellt wird. Die Grösse des darzustellenden Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für H kann entweder eine der Profan-Bitmaps (%HDC, %HDC2...) angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Ist H = 0, dann wird das Antialiasing zwar berechnet, aber noch nicht angezeigt.
Die Daten im Byte-Array B werden durch Aufruf von Antialiasing natürlich verändert.
R ist die transparente Farbe, an deren Grenzen die Kantenglättung erfolgt. Ist R = -1, dann wird die Farbe des rechten oberen Pixels von B als transparente Farbe genommen.
F sind diverse Flags. Ist F = 0, dann wird Methode 1 für das Antialiasing verwendet. F = 1 verwendet eine andere Methode. Zusätzlich kann noch Bit 1 von F gesetzt sein (also zu F den Wert 2 addieren), wenn nicht nur die Kanten, sondern die komplette Grafik geglättet werden soll. In dem Fall ist der Aufwand diese Funktion nicht so hoch und sie arbeitet schneller.
Ist A entweder 0, oder 1, dann wird Antialiasing() nur einmal durchlaufen. Es kann aber auch ein grösserer Wert angegeben werden, wenn die Kantenglättung stärker ausfallen soll. In dem Fall wird die Funktion öfter durchlaufen und dauert dadurch natürlich auch länger.

GetVersion ()

Ermittelt die vorliegende ProSpeed.dll Version.

Ergebniss: Long - Aktuelle Versioninfo

Das Ergebnis ist immer eine vierstellige Zahl !
Wird beispielsweise 2702 ausgegeben, dann bedeutet das: Version 2.7 Release 0.2, also 2.7.0.2.

SetCollideUnknownSprite (H1,H2,H3,H4)

Eine Spezifizierung zu CollideUnknown(). Wenn dort sehr viele Kollisionen erwartet werden müssen, kann die Anzahl mittels SetCollideUnknownSprite (auf ein bis vier Sprites) eingegrenzt werden.

```
H1 : Long - Ein Handle eines mit InitSprite() erzeugten Sprites oder 0
H2 : Long - Ein Handle eines mit InitSprite() erzeugten Sprites oder 0
H3 : Long - Ein Handle eines mit InitSprite() erzeugten Sprites oder 0
H4 : Long - Ein Handle eines mit InitSprite() erzeugten Sprites oder 0
```

CollideUnknown() erzeugt eine Liste aller momentan stattfindenen Kollisionen.
Manchmal interessieren aber nur die Kollisionen eines oder mehrerer Hauptsprites. Kollisionen von "Gegnersprites" untereinander (z.B. Wasserelemente oder Sprites, die als Vordergrundüberlagerung verwendet werden) interessieren nicht immer und eine aufwendige Kollisions-Analyse würde den Spielfluß stören.
In diesem Fall kann ein oder mehrere Sprites bestimmt werden, die bei einer Sprite-Sprite Kollision beteiligt sein müssen, wenn sie von CollideUnknown() wahrgenommen werden sollen. Alle anderen Kollisionen werden ausgefiltert.
Wird SetCollideUnknownSprite(0,0,0,0) übergeben, dann arbeitet CollideUnknown() wieder wie gewohnt.

```
SetCollideUnknownSprite(held&,waffe&,0,0)
```

AttachSpriteAnim (H,ZH)

Die Erweiterung zu AttachSprite(). Sprite H versucht automatisch genauso auszusehen wie Sprite ZH.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
ZH: Long - Ziel-Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0

Sehr geeignet z.B. für Jump&Run Spiele. Wenn sich in einem Level mehrere gleich aussehende Gegner befinden, können diese sehr einfach mittels AttachSprite() / AttachSpriteAnim() realisiert werden.
Um das Attach-Anim für Sprite H wieder aufzuheben, muß für ZH eine Null angegeben werden:

```
AttachSpriteAnim(attachsprite&,sprite&)
...
AttachSpriteAnim(attachsprite&,0) 'Attach-Anim-Modus wieder beenden
```

DSoundGetStatus (B)

Diese Funktion ermittelt den Spiel-Status eines Sounds.

```
B : Long - Zeiger auf einen Buffer, der von DSoundPlay() oder
DSoundGetNextBuffer() übergeben wurde.
```

Ergebniss: Long - Status

Das Ergebnis liefert den Status.
Ist Bit 0 (Wert 1) gesetzt, dann wird der Sound/Buffer gerade abgespielt, ansonsten ist er schon beendet.
Wenn Bit 2 (Wert 4) gesetzt ist, dann wird der Sound als Loop (Endlos) ausgegeben.

```
status&=DSoundGetStatus(playbuffer&)
```

VisualTableFix (B,A,L,O,R,U)

Setzt bei allen Sprites, die sich gerade außerhalb des sichtbaren Bereichs des Bildschirms befinden, den Fixmode 1. Diese Sprites werden also nicht mehr ständig neu gezeichnet.
Alle sichtbaren Sprites werden wieder in den Fixmodus 0 gesetzt, werden also immer neu gezeichnet.
VisualTableFix spart also, gerade bei riesigen Leveln, Unmengen an Systemzeit !

```
B : Long - Zeiger auf einen Speicherbereich, gefüllt mit Spritehandles
A : Long - Anzahl Bytes(!), mit denen der Speicher B gefüllt wurde
L : Long - Linker Rand
O : Long - Oberer Rand
R : Long - Rechter Rand
U : Long - Unterer Rand
```

L, O, R und U kennzeichnen den sichtbaren Rand eines Levels. Angenommen der Level ist 2000 x 2000 Pixel groß, der Bildschirm aber nur 800 x 600, dann wären die Werte für die vier Ränder: 0+Offset, 0+Offset, 800+Offset und 600+Offset.
B ist ein Speicher. Dieser wird mit den Spritehandles gefüllt, die von dem Wechsel des Fixmodes betroffen sein sollen.

A kennzeichnet, wieviele Bytes sich in B befinden (bitte beachten, ein Spritehandle belegt 4 Bytes (Long-Integer)...).
Mehr zum Fixmode gibt es unter GetSpriteFixMode() und SetSpriteFixMode() zu lesen.

```
VisualTableFix(bereich#,bytes&,scrollx&,scrolly&,scrollx&+800,scrolly&+600)
```

SlowDownSprite (H,X,Y)

Verlangsamt die Bewegung eines Sprite.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
X : Long - Verlangsamung der X-Achse
Y : Long - Verlangsamung der Y-Achse
```

Manchmal benötigt man Sprites, die sich langsamer als ein Pixel pro Frame bewegen sollen. Das ging bisher nicht (ohne Tricks).
Mit SlowDownSprite() ist das jetzt aber kein Problem mehr.
Jetzt wartet das Sprite H X, bzw Y Frames, bis das Sprite wieder in diese Richtung bewegt wird.
Um ein Sprite z.B. pro Frame um einen halben Pixel (rechnerisch, nicht visuell. In Wirklichkeit wird es einmal um einen Pixel bewegt, beim nächsten mal um 0 Pixel usw.) zu bewegen, muß geschrieben werden:

```
SetSpriteSpeed(sprite&,1,1)
SlowDownSprite(sprite&,1,1)
```

Wird für X oder Y eine 0 eingesetzt, so wird der ursprüngliche Zustand dieser Achse wieder hergestellt.

SaveImage (H,S,F,P)

Speichert ein Bild (HDC).

```
H : Long - HDC des Bildes / einer Bitmap
S : Long - Bereich mit dem Dateinamen oder Zeiger auf einen String mit dem
Dateinamen
F : Long - Flag, welches Speicherformat verwendet werden soll
P : Long - Parameter für diverse Speicherverfahren
```

Ergebnis: Long - 1, bei Fehler 0.

Das Bild mit dem HDC H wird unter dem Dateinamen S gespeichert.
F wählt das Format, in dem das Bild gespeichert werden soll.
Bisher gibt es folgende Werte für F:

```
0 = Bmp-Format
1 = Jpg-Format
```

Wenn F = 1 ist (speichern als .jpg), dann kann mit P die Qualität des Bildes bestimmt werden. Hierbei gelten Werte von 1 bis 10.
Je höher der Wert in P ist, desto besser ist die Qualität. Ich empfehle den Wert 8 zu benutzen.
Ist F = 0 (speichern als .bmp), dann hat P keine Bedeutung und sollte 0 sein.

Die Jpg-Speicherung basiert auf einer Dll von Andreas Miethe, geschrieben in PureBasic. Sie wurde mir freundlicherweise zur Verfügung gestellt. Danke, Andreas (Mail: andreas@andreas-miethe.de / Webpage: http://www.ampsoft.de/index.htm) !!!

Examine (Z)

Ermittelt, ob ein String eine Zahl enthält oder Text.

```
Z : Long - Zeiger auf einen String
```

Ergebnis: Long - 0 = Zahl / 1 = Text

Ein String (muß mit einem Nullbyte abschließen, ist bei Profan-Strings immer so) wird analysiert. Enthält der String nur die Zeichen "0123456789,.- ", dann wird er für eine Zahl gehalten.
Diese Methode funktioniert also auch bei negativen Zahlen und Dezimalzahlen !

```
text$="-1234,56"
Examine(addr(text$))
```

Smooth (F,X,Y,B,M)

Verwischt die Grafik, vergleichbar mit Blur().
Das Verwischen ist aber noch etwas softer und durch eine Maskenbitmap können die Passagen innerhalb des Bildes bestimmt werden, die vom dem Effekt betroffen sind. Natürlich langsamer als Blur()...

```
F : Long - Handle der Zielbitmap (HDC)
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quell/Zielbitmap)
M : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Maskenbitmap)
```

Ergebniss: Long - 0.

Die Grafiken in B wird unter Berücksichtigung der Maskenbitmap M verwischt und nach F und B "kopiert".
X und Y geben die linke obere Ecke an, an die beide Bilder gesetzt werden. Die Grösse des einzublendenen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Die Grösse von B und M muß aber immer gleich groß sein !!!
Für F kann entweder eine der Profan-Bitmaps (%HDC, %HDC2...) angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Ist F = 0, dann wird das Verwischen zwar weiter berechnet, aber nicht angezeigt.
Die Daten im Byte-Array M werden durch Aufruf von Smooth() nicht verändert, die Daten in B aber schon.
Die Maskenbitmap M sollte ein schwarz-weiß-Bild sein.
Weiße Pixel in der Maskenbitmap teilen Smooth() mit, das in diesen Bereichen der Verwisch-Effekt erzeugt werden soll. Schwarze Pixel in der Maskenbitmap bedeuten, das in diesen Bildbereichen kein Verwischen erzeugt wird.

Fog (F,X,Y,B1,B2,M)

Ein Nebeleffekt. Hierbei werden durch eine Maskenbitmap die Passagen innerhalb des Bildes bestimmt, die vom dem Nebeleffekt betroffen sind.

```
F : Long - Handle der Zielbitmap (HDC)
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B1: Long - Erstes Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
```

```
Bytes-Arrays (Quellbitmap)
B2: Long - Zweites Handle eines mit PngToFX(), InitFX() oder CreateFX()
erzeugten Bytes-Arrays (Zielbitmap)
M : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Maskenbitmap)
```

Ergebniss: Long - 0.

Die Grafiken der Bilder B1 und B2 werden unter Berücksichtigung der Maskenbitmap M mittels eines Nebeleffekts nach F und B2 kombiniert.
X und Y geben die linke obere Ecke an, an die beide Bilder gesetzt werden. Die Grösse des einzublendenen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Die Grösse von B1, B2 und M muß aber immer gleich groß sein !!!
Für F kann entweder eine der Profan-Bitmaps (%HDC, %HDC2...) angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Ist F = 0, dann wird der Nebel zwar weiter berechnet, aber nicht angezeigt.
Die Daten in den Byte-Arrays B1, M werden durch Aufruf von Fog() nicht verändert, die Daten in B2 aber schon.
Die Maskenbitmap M sollte ein schwarz-weiß-Bild sein.
Weiße Pixel in der Maskenbitmap teilen Fog() mit, das in diesen Bereichen der Nebel erzeugt werden soll.
Schwarze Pixel in der Maskenbitmap bedeuten, das in diesen Bildbereichen kein Nebel erzeugt wird.
Diese Funktion muß kontinuiertlich aufgerufen werden, um den Nebeleffekt richtig sichtbar werden zu lassen.

DeleteTags (Q,A,Z,C1,C2,F)

Entfernt, bzw. sammelt Text- oder Speicherpassagen, die sich innerhalb von zwei Tag-Zeichen befinden.

```
Q : Long - Zeiger auf einen Speicherbereich (oder String) mit den Quelldaten
A : Long - Anzahl Bytes, die in Q bearbeitet werden sollen
Z : Long - Zeiger auf einen Speicherbereich (oder String), in den die Zieldaten
geschrieben werden
C1: Long - ANSI-Code des Startbytes
C2: Long - ANSI-Code des Endbytes
F : Long - Flags
```

Ergebnis: Long - Anzahl Zeichen, die in Z generiert wurden.

Eine ziemlich vielseitige Funktion, die dem Anwender viel Arbeit abnehmen kann.
Z.B. könnten hiermit Html-Tags gelöscht werden, die in spitzen Klammern (< und >) stehen, oder Kommentare aus Quelltexten gelöscht werden (' und ANSI-Code 10), usw. Sicherlich finden sich noch allerhand individuelle Anwendungsgebiete...
Q und Z sind Zeiger auf Speicherbereich oder Strings.
C1 und C2 können Werte von 0 bis 255 sein, nicht höher! Beides sind ANSI-Code-Werte. Sie sollten immer unterschiedliche Werte erhalten.
Flags (F) gibt es bisher 2 (kombinierbar):

```
0 = kein Flag gesetzt.
1 = Nicht die Tags werden entfernt, sondern alles andere wird entfernt, was
nicht zwischen C1 und C2 liegt (Tags werden also gesammelt).
2 = Die Tag-Zeichen C1 und C2 sind vom Entfernen/Sammeln nicht betroffen
```

Das Ergebnis sind die Anzahl Zeichen/Bytes, die im Zielbereich Z generiert wurden. Zusätzlich wird als Endekennung ein Nullbyte eingefügt !
Beispiel siehe Quellcode *FA_DeleteTags.prf*.

BringSpriteToTop (H)

Setzt ein Sprite in der Darstellungs-Ebene (Layer) ganz nach oben.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Sprite H wird gegenüber allen anderen Sprites in den Vordergrund gestellt und kann somit von keinem anderen Sprite mehr überdeckt werden.
Nie auf das zuerst generierte Sprite anwenden !

BringSpriteToBottom (H)

Setzt ein Sprite in der Darstellungs-Ebene (Layer) ganz nach unten.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Sprite H wird gegenüber allen anderen Sprites in den Hintergrund gestellt und kann somit von jedem anderen Sprite überdeckt werden.
Eine Ausnahme ist das zuerst generierte Sprite, das immer auf der untersten Darstellungs-Ebene bleiben muß.

SetSpriteLayer (H,L)

Setzt für ein Sprite eine neue Darstellungs-Ebene (Layer).

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
L : Long - Neue Layer Nummer
```

Ergebniss: Long - 0 oder größer, bei Fehler -1

Der Layer des Sprites H wird neu vergeben. Je größer L ist, desto weiter wird das Sprite H gegenüber anderen Sprites in den Vordergrund gestellt.
L darf nicht kleiner sein als 1. Und L darf nicht größer sein als der Wert, der mit CountAllSprites() ermittelt werden kann, also der maximalen Anzahl vorhandener Sprites.
Wichtig! Das Sprite, welches als erstes generiert worden ist, darf und kann keine andere Darstellungs-Ebene bekommen. Es bleibt immer das Sprite, das von allen anderen überlagert werden kann! Es besitzt einen Sonderstatus.

```
SetSpriteLayer(sprite6&,3)
```

Pack (Q,Z,A,P)

Komprimiert (packt) einen Datenbereich per LZ77-Technik. Zusätzlich kann der gepackte Speicher mit einem Passwort geschützt werden.

```
Q : Long - Zeiger auf die Quelle der Daten, z.B. ein Zeiger auf einen Bereich
mit Daten.
Z : Long - Zeiger auf das Ziel (Zeiger auf einen Bereich), in dem die
komprimierten Daten abgelegt
     werden (Z darf hier nicht = Q sein, ein zweiter Speicherbereich ist nötig).
A : Long - Grösse des Datenbereichs (Anzahl zu packender Bytes).
P : Long - Zeiger auf einen String / Bereich, der ein Passwort enthält, oder 0
```

```
(kein Passwort verwenden)
```

Ergebniss: Long - Grösse des gepackten Datenbereichs (Anzahl gepackter Bytes, die jetzt in Z stehen). Bei Fehler = 0.

Achtung:
Die Technik, die hinter Pack() steht, stammt nicht aus meiner Feder. Es wird hier die aPLib von Joergen Ibsen verwendet, eine Library, die eine LZ komprimierende Technik verwendet, mehr dazu unten.
Wenn sie die ProSpeed.dll in Freewareprogramm benutzen, dürfen sie Pack() / aPLib kostenlos benutzen, aber in Shareware Programmen und kommerizieller Software müssen sie aPLib registrieren.
Die Lizenz für aPLib erhalten sie für $ 29 (Firmen für $ 95) bei Joergen Ibsen (Webpage / Email).
Mit freundlicher Genehmigung. aPLib, copyright © 1998-2005 by Joergen Ibsen.

Seit ProSpeed 2.6 gibt es diese Packfunktion. Pack() ist ein wahrer Alleskönner, aber leider recht langsam.
Pack() komprimiert dafür etwa gleich gut oder besser als WinZip, hier mal die Pack-Ergebnisse verschiedener Verfahren auf eine über 3 MB große Textdatei (Sieger Winrar ist hier unschlagbar):

```
Originaler Text                   = 3.087.250 Bytes (100,0 %)
mit WinZip (beste Stufe) gepackt  =   462.998 Bytes  (15,0 %)
mit Pack() gepackt                =   446.721 Bytes  (14,5 %)
mit WinRar gepackt                =   184.458 Bytes   (6,0 %)
```

Pack() kann also sehr gut mithalten, im Vergleich zum (momentanen) Marktführer Winzip. Packraten von über 80% sind darum keine Seltenheit.
Hier noch ein zweites Beispiel, diesmal ein riesiges Bmp-Bild (4096 x 4096 !) mit über 50 MB belegtem Speicher.

```
Originales Bild (.bmp)            = 50.331.702 Bytes (100,0 %)
als Bild (.png)                   =    661.447 Bytes   (1,3 %)
mit WinZip (beste Stufe) gepackt  =    611.139 Bytes   (1,2 %)
mit Pack() gepackt                =    575.345 Bytes   (1,1 %)
mit WinRar gepackt                =    382.297 Bytes   (0,8 %)
```

Die Reihenfolge hat sich nicht geändert. PNG ist hier noch mit dabei, die beste verlustfreie Komprimierungs-Methode für Grafiken.
Wow, Pack() mit fast 99 % Platzersparniss, das kann sich schon sehen lassen...

Q und Z sind Zeiger auf Speicherbereiche, in Q stehen die Daten, die gepackt werden sollen, in Z werden die komprimierten Daten dann zurück geschrieben. Wichtig: Es müssen zwei verschiedene Speicherbereiche benutzt werden !
Der Speicher in Z muß genug Platz enthalten, um im Falle einer erfolglosen Komprimierung keinen Fehler zu verursachen. Mit dieser Formel kann die mindest Anzahl Bytes innerhalb von Z berechnet werden:

```
Anzahl Bytes = A + (A / 8) + 64
```

A gibt an, wieviele Daten in Q gepackt werden sollen.
P kann ein String sein (oder auch ein Bereich), in dem ein Passwort steht. Ist P = 0, dann werden die Daten nur gepackt. Steht in P aber ein Passwort, dann werden die gepackten Daten zusätzlich noch mit diesem Passwort verschlüsselt. Die Verschlüsselung basiert auf der Funktion Crypt(), dort gibt es mehr Infos hierüber.
Mit Pack() komprimierte Daten können nur mit DePack() wieder dekomprimiert / entschlüsselt werden !
Und Achtung: Eine gepackte Datei kann ohne das gültige Passwort nicht mehr entpackt werden !

Hier ein Beispiel, um Daten mit Pack() zu laden, komprimieren und zu verschlüsseln:

```
text$="Dateiname"
bytes&=FileSize(text$)

Dim bereich#,bytes&
Dim zielbereich#,bytes&+(bytes&/8)+64

ReadFileFast(addr(text$),bereich#,0,bytes&)

text$="Passwort"
x&=Pack(bereich#,zielbereich#,bytes&,addr(text$))

Print "Alte Dateilänge ="+Str$(bytes&)+" Bytes."
Print "Neue Dateilänge ="+Str$(x&)+" Bytes ("+Str$(x&*100/bytes&)+" % übrig)."
```

```
AlteDateilänge= 48030 Bytes.
Neue Dateilänge= 5408 Bytes (11,259629% übrig).
```

DePack (Q,Z,P)

Dekomprimiert (entpackt) einen Datenbereich wieder, der mit Pack() komprimiert und / oder verschlüsselt wurde.

```
Q : Long - Zeiger auf eine Quelle (Zeiger auf einen Bereich oder String) mit den
gepackten Daten.
Z : Long - Zeiger auf das Ziel, in dem die dekomprimierten Daten abgelegt werden
     (Z darf nicht = Q sein, ein zweiter Speicherbereich ist nötig).
P : Long - Zeiger auf einen String / Bereich, der ein Passwort enthält, oder 0
(kein Passwort verwenden)
```

Ergebniss: Long - Anzahl entpackter Bytes. Bei Fehler = 0.

Die Technik, die hinter DePack() steht, stammt nicht aus meiner Feder. Es wird hier die aPLib von Joergen Ibsen verwendet, eine Library, die eine LZ komprimierende Technik verwendet, mehr dazu unten.
Wenn sie die ProSpeed.dll in Freewareprogramm benutzen, dürfen sie DePack() / aPLib kostenlos benutzen, aber in Shareware Programmen und kommerizieller Software müssen sie aPLib registrieren.
Die Lizenz für aPLib erhalten sie für $ 29 (Firmen für $ 95) bei Joergen Ibsen (Webpage / Email).
Mit freundlicher Genehmigung. aPLib, copyright © 1998-2002 by Joergen Ibsen.

Entpackt Daten wieder, die mit Pack() gepackt wurden
Es ist nicht nötig, die Anzahl gepackter Bytes irgendwo anzugeben, da diese Information in die gepackten Daten integriert wurde.
Um herauszufinden, wieviele Bytes die entpackten Daten nachher belegen, muß GetOriginalSize() verwendet werden.
DePack() arbeitet äußerst schnell.

Beispiel (zum Laden, Entpacken und Entschlüsseln, siehe auch Beispiel von Pack()):

```
text$="Dateiname"
bytes&=FileSize(text$)

Dim bereich#,bytes&
ReadFileFast(addr(text$),bereich#,0,bytes&)

neubytes&=GetOriginalSize(bereich#)
```

```
Dim zielbereich#,neubytes&

Print "Daten gepackt = "+Str$(bytes&)+" Bytes."
Print "Errechnete, zu entpackende Daten = "+Str$(neubytes&)+" Bytes."

text$="Passwort"
x&=DePack(bereich#,zielbereich#,addr(text$))
Print
Print "Daten entpackt = "+Str$(x&)+" Bytes."
```

Daten gepackt= 5408 Bytes.
Errechnete,zu entpackende Daten = 48030 Bytes.

Daten entpackt= 48030 Bytes..

GetOriginalSize (Q)

Ermittelt die Anzahl entpackter Bytes, die entstehen werden, wenn ein mit Pack() gepackten Speicherbereich entpackt wird.

```
Q : Long - Zeiger auf eine Quelle (Zeiger auf einen Bereich oder String) mit den
gepackten Daten.
```

Ergebniss: Long - Anzahl entpackter Bytes (originale Länge). Bei Fehler = 0.

Diese Funktion kann nur auf Daten angewendet werden, die mit Pack() komprimiert wurden.
Sinnvoll kann sie sein, um zu erfahren, wieviel Speicher für DePack() reserviert werden muß.

```
bytes&=GetOriginalSize(bereich#)
```

DSoundLoadMemory (B,A)

Lädt eine Wave-Datei aus dem Arbeitsspeicher zum späteren Abspielen.

```
B : Bereichsvariable
A : Long - Anzahl Bytes im Bereich
```

Ergebniss: Long - 0 = Fehler / ansonsten Zeiger auf ein Buffer-Array (wichtig für DSoundUnLoad()).

Eine Wav-Datei wird aus einem Speicherbereich geladen und in einem Buffer-Array gespeichert.
Eigentlich werden in dem Buffer-Array 15 Kopien des Sound angelegt, plus eine Zählervariable. Diese 15 Kopien belegen intern aber denselben DirectSound-Speicher, sodaß kein Speicher vergeudet wird.
Warum 15 Kopien? Jede Kopie kann nur einmal zur gleichen Zeit abgespielt werden. Wenn wir aber 15 Kopien haben, kann auch 15 mal ein und dergleiche Sound gleichzeitig abgespielt werden, und das sollte für jedes Spiel reichen, was wir entwerfen...

(Diese Funktion besitzt keinen Extra-Offsetparameter, wie bei vielen Bereichsfunktionen eigentlich üblich. Ob jetzt ein Komma geschrieben wird, oder ein Pluszeichen, ist eigenlich egal, wie folgendes Demo beweist, was vier Samples aus einem Speicherbereich lädt.)

```
buffer1&=DSoundLoadMemory(bereich#,8200)
buffer2&=DSoundLoadMemory(bereich#+8200,16003)
buffer3&=DSoundLoadMemory(bereich#+24203,4004)
buffer4&=DSoundLoadMemory(bereich#+28207,22401)
```

DSoundLoadResource (H,S)

Lädt eine Wave-Datei aus einer Resource-Datei zum späteren Abspielen. Eine Resource-Datei kann eine Dll sein oder ein Exeprogramm (z.B. das eigene), das mit Daten zusammen gelinkt wird.

```
H : Long - HInstance einer Exedatei oder das Handle einer geladenen Dll.
S : Long - Nummer der Resource oder String mit Namen der Resource
```

Ergebniss: Long - 0 = Fehler / ansonsten Zeiger auf ein Buffer-Array (wichtig für DSoundUnLoad()).

Eine Wav-Datei wird als Resource geladen und in einem Buffer-Array gespeichert.
Eigentlich werden in dem Buffer-Array 15 Kopien des Sound angelegt, plus eine Zählervariable. Diese 15 Kopien belegen intern aber denselben DirectSound-Speicher, sodaß kein Speicher vergeudet wird.
Warum 15 Kopien? Jede Kopie kann nur einmal zur gleichen Zeit abgespielt werden. Wenn wir aber 15 Kopien haben, kann auch 15 mal ein und dergleiche Sound gleichzeitig abgespielt werden, und das sollte für jedes Spiel reichen, was wir entwerfen...

Für S kann eine Zahl (Long) oder ein String (Resourcename) angegeben werden. Es kann sogar eine Zahl im String angegeben werden, dann muß das erste Zeichen im String ein # sein und der Rest wird als Zahl interpretiert, z.B. text$="#100". Wird für S ein String angegeben, dann muß er mindestens 3 Zeichen enthalten, Namen mit weniger Zeichen sind nicht gestattet.
Für H kann eine Dll oder der HInstance des Exeprogramm angegeben werden, auch ein fremdes Exeprogramm, das Resourcen enthält, kann angegeben werden, sofern der HInstance dieses Programms ermittelt wurde.
Wichtig: Wave-Dateien in Resourcen müssen als Resourcetyp "WAVE" gespeichert sein !

Beispiele:

1.)

```
DSoundInit(%hwnd)
dll&=usedll("Resource.dll")

text$="SOUND1"
buffer1&=DSoundLoadResource(dll&,addr(text$))
```

2.)

```
buffer1&=DSoundLoadResource(%HInstance,100)
```

ExchangeColor (F,X,Y,B,Q,Z)

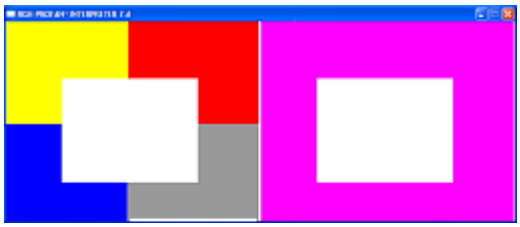
Tauscht eine Farbe einer Grafik (alle Pixel mit dieser Farbe) durch eine andere Farbe.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
Q : Long - Quellfarbe
Z : Long - Zielfarbe
```

Ergebniss: Long - 0.

Im Bytearray B werden alle Pixel mit der Quellfarbe Q umgefärbt mit Farbe Z.
X und Y geben die linke obere Ecke an, an die die Grafik B nach dem Umfärben nach F kopiert wird. Die Grösse des neu zu färbenden Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird das Umfärben zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
ExchangeColor() ist ziemlich schnell und arbeitet in Echtzeit.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

Beispiel:
(ein Testbild wird geladen (mit den Farben gelb, rot, blau, grau und weiss) und in die linke Fensterhälfte kopiert. Dann werden vier Farben des Bilds (gelb, rot, blau und grau) violett eingefärbt und in die rechte Fensterhälfte kopiert.)



```
text$="Test.bmp"
hdc&=LoadFileImage(addr(text$))

PaintImage(%hdc,0,0,444,356,hdc&,0,0,0)

array&=InitFX(hdc&)

ExchangeColor(0,0,0,array&,Rgb(255,255,0),Rgb(255,0,255)) 'gelb nach violett
ExchangeColor(0,0,0,array&,Rgb(255,0,0),Rgb(255,0,255))   'rot nach violett
ExchangeColor(0,0,0,array&,Rgb(0,0,255),Rgb(255,0,255))   'blau nach violett
ExchangeColor(%hdc,448,0,array&,Rgb(153,153,153),Rgb(255,0,255)) 'grau nach
violett und Bild kopieren nach %hdc, X=448, Y=0
FreeFX(array&)
FreeImage(hdc&)
```

RotateMem (B,O,A,R,F)

Eine Funktion, um Daten in einem Bereich (oder String) rotieren zu lassen.

```
B : Long - Zeiger auf einen Speicherbereich (oder String)
O : Long - Offset (Adresse) für B
A : Long - Anzahl Bytes die rotiert werden sollen
R : Long - Rotieren um so viele Bytes
F : Long - Flag, 0 = nach rechts rotieren / 1 = nach links rotieren
```

Ergebnis: Long - 0

Hiermit können Daten in einem Speicherbereich nach links oder rechts verschoben werden. Daten, die hinten (bzw. vorne) aus dem angegebenen Speicher "herausfallen", werden vorne (bzw. hinten) wieder angehängt.
Am besten demonstrieren das einige Beispiele:

Beispiel 1:

```
Clear bereich#
Long bereich#,0=111
Long bereich#,4=222
Long bereich#,8=333
Long bereich#,12=444
Long bereich#,16=555

RotateMem(bereich#,0,20,4,0)

Print "Vier Bytes nach rechts rotieren !"
Print Long(bereich#,0)
Print Long(bereich#,4)
Print Long(bereich#,8)
Print Long(bereich#,12)
Print Long(bereich#,16)
```

```
Vier Bytes nach rechts rotieren !
555
111
222
333
444
```

Beispiel 2:

```
Clear bereich#
Long bereich#,0=111
Long bereich#,4=222
Long bereich#,8=333
Long bereich#,12=444
Long bereich#,16=555

RotateMem(bereich#,0,20,4,0)

Print "Jetzt vier Bytes nach links rotieren !"
Print Long(bereich#,0)
Print Long(bereich#,4)
Print Long(bereich#,8)
Print Long(bereich#,12)
Print Long(bereich#,16)
```

```
Jetzt vier Bytes nach links rotieren !
222
333
444
555
111
```

Beispiel 3:
(Der Inhalt eines Strings wird um 10 Buchstaben (Bytes) nach rechts rotiert, sehr schön, um ein Textscrolling zu programmieren !)

```
text$="Dieser Text ist ein Probetext."
```

```
RotateMem(addr(text$),0,len(text$),10,0)
Print text$
```

Probetext.DieserTextistein

Frame (B,H,L,X1,Y1,X2,Y2,W)

Eine Funktion, um schnell einen 3D-Rahmen (mit variabler Strichbreite) zu zeichnen.

```
B  : Long - Handle des HDC's, in den gemalt werden soll
H  : Long - Hightlight Farbe (Rahmenbalken links und oben)
L  : Long - Lowlight Farbe (Rahmenbalken rechts und unten)
X1 : Long - Koordinate der linken Ecke
Y1 : Long - Koordinate der oberen Ecke
X2 : Long - Koordinate der rechten Ecke
Y2 : Long - Koordinate der unteren Ecke
W  : Long - Breite des Rahmens in Pixel
```

Ergebnis: Long - 0

Hiermit kann schnell ein 3D-Rahmen auf einen HDC gebracht werden.

Beispiel, um einen 4 Pixel breiten Rahmen ins Profan-Hauptfenster zu zeichnen, in den Farben schwarz und weiß:

```
Frame(%HDC,Rgb(0,0,0),Rgb(255,255,255),30,40,100,200,4)   'ins Hauptfenster
Frame(%HDC2,Rgb(0,0,0),Rgb(255,255,255),30,40,100,200,4)  'und einmal in Profans
Sicherheitskopie
```

CollideUnknown (B)

Ermittelt in einem Rutsch alle (!) Sprite-zu-Sprite Kollisionen, die momentan stattfinden !

```
B : Long - Zeiger auf einen Bereich, der die Kollisionsdaten aufnehmen kann
```

Ergebnis: Long - Anzahl gefundener Sprite-Kollisionen.

Lange gefordert, jetzt gibt es sie endlich! Eine Komplettlösung für alle Sprite-Kollisionen, die gerade stattfinden.
CollideUnknown() arbeitet innerhalb einer Millisekunde und listet alle Kollisionen im Bereich B auf. Zusammen mit MarkSprite() lassen sich so alle Sprites sinnvoll und schnell einordnen.
Jeweils zwei Spritehandle (zu einer Kollision gehören ja immer zwei Sprites) werden als LongInt hintereinander in den Bereich B geschrieben.

```
Offset   Bedeutung
0        1. Sprite-Handle von Kollision 1
4        2. Sprite-Handle von Kollision 1
8        1. Sprite-Handle von Kollision 2
12       2. Sprite-Handle von Kollision 2
16       1. Sprite-Handle von Kollision 3
20       2. Sprite-Handle von Kollision 3
...
```

Kleines Programmbeispiel, um diese Kollisionen auszuwerten:

```
Clear kollis#
x&=CollideUnknown(kollis#)
If x&
 y&=0
 Whileloop x&
  a&=Long(kollis#,y&)
  b&=Long(kollis#,y&+4)
  Print "Kollision Nr.: "+Str$(x&)
  Print "Spritehandle 1 = "+Str$(a&)
  Print "Spritehandle 2 = "+Str$(b&)
  ' *** Hier könnte jetzt auf bestimmte Kollisionen entsprechend
  ' *** reagiert werden (z.B. via MarkSprite)
  y&=y&+8
 Wend
EndIf
```

Es sollte noch erwähnt werden, das doppelte Kollisionen von ProSpeed schon ausgefiltert werden, das heißt also, das zwei gleiche Handle nicht zweimal in unterschiedlicher Reihenfolge vorkommen.
Das heißt aber nicht, das, wenn sich mehr als zwei Sprites berühren, diese nicht aufgeführt werden !
Hier noch ein Beispiel - angenommen es gäbe drei Sprites und alle drei Sprites würden sich untereinander berühren. Dann sähe der Inhalt von B so aus:

```
 0       Handle von Sprite 1 \
               )- Kollision 1
 4       Handle von Sprite 2 /

 8       Handle von Sprite 1 \
               )- Kollision 2
12        Handle von Sprite 3 /

16        Handle von Sprite 2 \
               )- Kollision 3
20        Handle von Sprite 3 /
```

CollideUnknown() gibt in diesem Fall einen Wert von 3 (also drei Kollisionen) zurück.

PMouse (X1,Y1,X2,Y2)

Ein Ersatz für Profan's Funktion Mouse(...), die nicht funktioniert, wenn die Usermessage WM_MOUSEMOVE gesetzt wurde. PMouse() funktioniert auch dann.

```
X1 : Long - Linke Ecke
Y1 : Long - Obere Ecke
X2 : Long - Rechte Ecke
Y2 : Long - Untere Ecke
```

Ergebnis: Long - 0 oder ungleich 0.

Befindet sich der Mauszeiger gerade innerhalb des angegebenen Rechtecks, ist das Ergebnis <> 0, wenn nicht, ist das Ergebnis 0.

GetInfosImage (H,B)

Ermittelt einige Informationen zu einer Bitmap, die mit LoadFileImage() geladen oder mit CreateExtImage() erzeugt wurde.

```
H : Long - Handle der Bitmap / des HDC's
B : Long - Zeiger auf einen Speicher, der die Informationen empfangen soll (40
Bytes).
```

Ergebnis: Long - Wenn alles glattging 1, bei Fehler 0.

Beispiel:

```
Dim bereich#,44
GetInfosImage(HDC&,bereich#)
Print "Strukturgröße = "+Str$(Long(bereich#,0))              '40, unwichtig
Print "Bitmap-Breite = "+Str$(Long(bereich#,4))
Print "Bitmap-Höhe = "+Str$(Long(bereich#,8))
Print "Anzahl Planes = "+Str$(Word(bereich#,12))             'immer 1
Print "Bit-Tiefe = "+Str$(Word(bereich#,14))
Print "Kompression = "+Str$(Long(bereich#,16))               'fast immer BI_RGB
Print "Speicherverbrauch der Bitmap = "+Str$(Long(bereich#,20))
'plus 54 Bytes Header = .bmp Größe
Print "XPelsPerMeter = "+Str$(Long(bereich#,24))             'unwichtig
Print "YPelsPerMeter = "+Str$(Long(bereich#,28))             'unwichtig
Print "ClrUsed = "+Str$(Long(bereich#,32))                   'eher unwichtig
Print "ClrImportant = "+Str$(Long(bereich#,36))              'eher unwichtig
```

ColorFillImage (V,R,X1,Y1,X2,Y2)

Füll einen rechteckigen Bereich einer Bitmap in V mit Farbe R.

```
V  : Long - HDC
R  : Long - RGB Farbwert.
X1 : Long - Linker Offset in Pixel, an dem die Füllung beginnt.
Y1 : Long - Oberer Offset in Pixel, an dem die Füllung beginnt.
X2 : Long - Rechter Offset in Pixel, an dem die Füllung endet.
Y2 : Long - unterer Offset in Pixel, an dem die Füllung endet.
```

Ergebnis: 0

Beispiel:

```
ColorFillImage(HDC&,RGB(255,0,0),0,0,%maxx,%maxy)     'Füllt   den   HDC   komplett
rot.
```

Manchmal muß man einen HDC mit einer bestimmten Farbe füllen. Hiermit ist das jetzt ohne großen Aufwand möglich. Aber ColorFillImage() ist trotzdem so flexibel gestaltet, das auch nur ein beliebiger rechteckiger Bereich ausgefüllt werden kann.

SetTiles5 (B)

Eine Profan 5 kompatible Lösung, um Tiles zu benutzen.

```
B : Long - Bereich, in dem die 15 Tiles-Parameter hintereinander (als Long-Int)
abgelegt sind
```

Ergebniss: 0

Profan 5 kann nur bis zu 12 Parameter an eine Dll übergeben, SetTiles() besitzt aber 15 Parameter. Darum ist SetTiles5() die einzige Möglichkeit, Tiles in "kleinen" Profanversionen zu benutzen.
Welche Parameter benötigt werden, ist bei SetTiles() nachzulesen.
Der Speicherbereich B muß mit mindestens 64 Bytes dimensioniert sein.

Ein Beispielsource ist bei den ProSpeed.dll Sourcecodes zu finden: **FA_Tiles_Profan5.prf**

DataMem (B)

Setzt einen neuen Bereich für DataBytes(). Das nächste DataBytes() greift dann auf diesen Bereich ab Position 0 zu.

```
B : Neuen Bereich für das nächste DataBytes() bestimmen und Byte-Offset auf 0 setzten
```

Ergebnis: Long - 0

Näheres hierzu unter DataBytes().

DataPos (P)

Setzt eine neue Position für DataBytes(). Das nächste DataBytes() greift dann ab dieser Position auf den Bereich zu.

```
P : Position (Offset) für das nächste DataBytes()
```

Ergebnis: Long - 0

Näheres hierzu unter DataBytes().

DataBytes (S)

Eine Implementierung des Basic-Befehls DATA für Profan. "DataBytes()" gibt es auch als Prozedur "Data".

```
S : Addresse eines Strings mit Eingabe-Instruktionen
```

Ergebnis: Long - 0

Profan kennt keinen DATA-Befehl. Praktisch wäre aber so eine Funktion, um einem Speicherbereich (Bereich) Daten zuzuweisen.
ProSpeed stellt hierzu schon seit längerem SetBytes() zur Verfügung. DataBytes / Data geht aber noch einen Schritt weiter.
Zuerst wird mit DataMem() bestimmt, auf welchen Bereich DataBytes / Data als nächstes zugreifen soll. DataMem() setzt hierzu automatisch die Positionierung auf das erste Byte des angesprochenen Bereichs (DataPos(0)):

```
DataMem (bereich#)
Data "250,128,23,56,251,83,90,12,7,252,253,0,26,124,78,78,43,202"
Data "1,254,59,255,77,200,201,45,202,203,204,205,206,207,56,23,78"
Data "228,23"
```

```
DataMem (speicher#)
Data "206,56,23,78"
Data "250,124,78,78"
Data "77,200,201,45"
```

(Dieser Quellcode verwendet die Prozedure "Data". Wenn sie in ihrem Programm die Datei *ProSpeed_Funktionen.inc* einbinden, können sie darauf zugreifen, ab Profan 7.)

Mit diesem Quellcode würden die ersten 37 Bytes des Bereichs "bereich#" beschrieben, danach die ersten 12 Bytes des Bereichs "speicher#".
Databytes() / Data kann sehr schön dafür verwendet werden, Bilder oder andere Daten direkt in den Quelltext einzubinden (um Daten kompakter einzubinden, benutzen sie bitte Oembytes() / Oem).
Der String S darf nur Ziffern, Kommata, Leerzeichen und Sternzeichen enthalten, Variablen sind nicht erlaubt. ProSpeed wandelt diesen String in die einzelnen Bytes um und schreibt diese in den Bereich. Das Sternzeichen ist ein Dummy-Zeichen. Mehr hier unter SetBytes().

Im ProSpeed Paket befindet sich das Programm **Datagenerator** (als Source und als Exe), mit dem sehr bequem eine beliebige Datei in Data-Zeilen umgewandelt werden kann. Die so generierten Zeilen können direkt in den Profan-Source kopiert werden, oder als Inc-Datei abgespeichert werden.
Ein anschauliches Beispiel bietet der Datagenerator-Quellcode selber. Auf jeden Fall mal reinschauen.

OemBytes (S)

Eine kompakte Form von DataBytes() / Data. Die Daten werden aber nicht als Zahlen übergeben, sondern als OEM-Zeichen.
"OemBytes()" gibt es auch als Prozedur "Oem".

```
S : Addresse eines Strings mit Eingabe-Instruktionen
```

Ergebnis: Long - 0

OemBytes / Oem geht noch einen Schritt weiter als DataBytes / Data.
Der Code in S ist wesentlich kompakter, dafür aber nicht mehr editierbar.
Zuerst wird mit DataMem() bestimmt, auf welchen Bereich OemBytes / Oem als nächstes zugreifen soll. DataMem() setzt hierzu automatisch die Positionierung auf das erste Byte des angesprochenen Bereichs (DataPos(0)):

```
Declare bereich#
Dim bereich#,733
Clear bereich#
DataMem(bereich#)
Oem "ÿØÿà # 3JFIF # $ $ # # $ # $ # #ÿÛ #C # . + + - + * . -"
Oem " , - 0 / . 0 4 ? 5 4 2 2 4 E < = 7 ?)$+*($''-2@7-0=0''8"
Oem "L9=CEHIH+6OUNFT@GHEÿÛ #C $ / 0 0 4 2 4 D 5 5 DE.'.EEEEE"
Oem "EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEÿÀ # 4 + #"
Oem "0 #: & $ E # % 4 $ & 4 $ÿÄ # B # # $ ( $ $ $ $ $ $ # # #"
Oem " # # # # # $ % & ' ( ) * + , - .ÿÄ #µ 3 # % $ & & % ' &"
Oem " ( ( ' ' # # $} $ % & # ' 4 ( 5 D1A ) 6Qa * Eq 72□'¡ +#"
Oem "B±Á 8RÑð$3br, , - 9 : ; < =%&'()*456789:CDEFGHIJSTUVWXY"
Oem "Zcdefghijstuvwxyzƒ„…†‡ˆ‰Š'"•!2!3˜™š$¢£¤¥¦§¨©ª²³´µ¶·¸¹°Â"
Oem "ÃÄÅÆÇÈÉÊÒÓÔÕÖ×ØÙÚáâãäåæçèéêñòóôõö÷øùúÿÄ # B $ # & $ $ $"
Oem " $ $ $ $ $ $ # # # # # # $ % & ' ( ) * + , - .ÿÄ #µ 4 #"
Oem " % $ % ' ' & ' * ( ' ' # $ %w # $ % & 4 ' ( D1 ) 5AQ *a"
Oem "q 6 E2□ + 7B'¡±Á ,#3Rð 8brÑ - 9$4á%ñ : ; < =&'()*56789:"
```

```
Oem "CDEFGHIJSTUVWXYZcdefghijstuvwxyz,ƒ„…†‡ˆ‰Š'"•!2!3˜™š¢£¤"
Oem "¥¦§¨©ª²³´µ¶·¸¹ºÂÃÄÅÆÇÈÉÊÒÓÔÕÖ×ØÙÚâãäåæçèéêòóôõö÷øùúÿÚ #"
Oem " / & $ # % 4 & 4 #? #òª(ïK^êM·© %QKE>_0 5Š^ÔQo0 -(ìhªµ'"
Oem " 3□é{Rw¥íR¾ÐÃ¸¢□J;ÕuW #íA£µ 7ŸP 1ÆŠ; =*žÈBw¥¢Ž=*TwÔaE ?"
Oem "zQÇ¥U¼Ä'jZ8ô£□J □«¸ □; =(Í 7ß@?ÿÙ"
```

(Dieser Quellcode verwendet die Prozedure "Oem". Wenn sie in ihrem Programm die Datei *ProSpeed_Funktionen.inc* einbinden, können sie darauf zugreifen, ab Profan 7.)

Mit diesem Quellcode würden 732 Bytes in den Bereich "bereich#" beschrieben, in diesem Fall ein kleines Jpg-Bild.
Das Beispiel wurde mit dem Programm **Oemgenerator** erstellt (liegt dem ProSpeed Paket als Source und als Exe bei), mit dem sehr bequem eine beliebige Datei in Oem-Zeilen umgewandelt werden kann. Die so generierten Zeilen können direkt in den Profan-Source kopiert werden, oder als Inc-Datei abgespeichert werden.
OemBytes() / Oem kann also sehr schön dafür verwendet werden, Bilder oder andere Daten direkt in den Quelltext einzubinden.
Ein anschauliches Beispiel bietet der Oemgenerator-Quellcode selber. Auf jeden Fall mal reinschauen.

AlphaTrans (F,X,Y,B1,B2,M)

Mischt zwei verschiedene Bilder transparent zu einem, hierbei wird durch eine Maskenbitmap der Transparenz-Grad für beide Bilder bestimmt.

```
F  : Long - Handle der Zielbitmap (HDC) oder 0
X  : Long - X Offset zum Bildschirm
Y  : Long - Y Offset zum Bildschirm
B1 : Long - Erstes Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays (1. Quellbitmap)
B2 : Long - Zweites Handle eines mit PngToFX(), InitFX() oder CreateFX()
erzeugten Bytes-Arrays (2. Quellbitmap)
M  : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays (Maskenbitmap)
```

Ergebniss: Long - 0.

Die Grafiken der Bilder B1 und B2 werden unter Berücksichtigung der Maskenbitmap M nach F kombiniert.
X und Y geben die linke obere Ecke an, an die beide Bilder gesetzt werden. Die Grösse des einzublendenen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Die Grösse von B1, B2 und M muß aber immer gleich groß sein !!!
Für F kann entweder eine der Profan-Bitmaps (%HDC, %HDC2...) angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F eine 0 angegeben, dann erfolgt die Ausgaben nicht in einen HDC, sondern in das Byte-Array B1!
Die Daten in den Byte-Arrays B1, B2 und M werden durch Aufruf von AlphaTrans() nicht verändert, es sei denn, für F wird 0 angegeben. Aber auch dann wird nur B1 verändert. Ansonsten arbeitet AlphaTrans intern mit einem vierten Speicherbereich.
Die Maskenbitmap M sollte ein schwarz-grau-weiß-Bild sein. Es lassen sich aber auch mit farbigen Maskenbitmaps tolle Effekte realisieren.
Je weißer ein Pixel in der Maskenbitmap ist, desto sichtbarer wird der Pixel von Quellbitmap 1 (B1). Je schwärzer ein Pixel in der Maskenbitmap ist, desto sichtbarer wird der Pixel von Quellbitmap 2 (B2).
Zwei Beispiele für AlphaTrans() befinden sich im ProSpeed Sourcecode-Ordner.
Siehe auch BitBltArray().

Tooltip (D,S,W,I)

Hiermit läßt sich zu einem Dialogelement (Button, Editfeld usw.) ein Tootip generieren.

"Tooltip()" gibts auch als Prozedur "SetTooltip".

```
D : Long - Handle eines Controls (Dialogelements)
S : Long - Addresse eines Strings mit Text für den Tooltip
W : Long - %hwnd (Handle des Hauptfensters)
I : Long - %hinstance
```

Ergebnis: Long - Handle des generierten Tooltip

Tooltips sind kleine Texte, die der Computer darstellt, sobald der Mauszeiger kurz über einem Dialogelement anhält.
Ein installierter Tooltip braucht zum Programmende nicht wieder freigegeben werden.

```
button&=@CreateButton(%hwnd,"Ende",200,312,80,20)
text$="Dieser Knopf beendet das Programm"
Tooltip(button&,addr(text$),%hwnd,%hinstance)
```

Wenn sie die Inc-Datei "Prospeed_Funktionen.inc" verwenden, bietet ProSpeed auch eine Tooltip-Prozedur an, weil Profan sowieso schon zwei der Parameter kennt, %hwnd und %hinstance.
In diesem Falle können sie einen Tooltip noch einfacher installieren, die Prozedur heißt "**SetTooltip**" (erst verwendbar ab Profan 7!):

```
button&=@CreateButton(%hwnd,"Ende",200,312,80,20)
SetTooltip button&,"Dieser Knopf beendet das Programm"
```

SwitchTaskbar (F)

Läßt Windows Taskbar verschwinden oder erscheinen.

```
F : Long - Flag
```

Ergebnis: Long - 0

Hiermit kann die Taskbar im Windowsfenster an- und ausgeschaltet werden.
Werte für F sind:

```
0 = Taskbar abschalten
1 = Taskbar anschalten
```

SetTiles (Q,YT,AX,AY,B,H,M,BM,MX,MY,Z,OX,OY,T,F)

Setzt ein Bild (bzw. Level) zusammen, das aus lauter kleinen Grafikteilchen (Tiles). Die Anordnung der Grafikteile wird durch eine Maptabelle bestimmt.

```
Q  : Long - HDC eines Bildes mit den verschiedenen Grafikplättchen
YT : Long - Y-Offset in Q, ab der die Grafiken liegen (gemessen in Pixel)
AX : Long - Anzahl Tiles, die auf der X-Achse kopiert werden sollen (nach Z)
AY : Long - Anzahl Tiles, die auf der Y-Achse kopiert werden sollen (nach Z)
B  : Long - Breite eines Tile
H  : Long - Höhe eines Tile
M  : Long - Zeiger auf einen Bereich mit den Mapdaten (z.B. map#)
BM : Long - Breite der Map in Tiles
MX : Long - X-Offset in den Mapdaten (M), ab wo die Tiles in Z dargestellt
werden sollen
```

```
MY : Long - Y-Offset in den Mapdaten (M), ab wo die Tiles in Z dargestellt
werden sollen
Z  : Long - Ziel HDC, in die alle betroffenen Tiles kopiert werden
OX : Long - X-Offset in der Ziel HDC (Z)
OY : Long - Y-Offset in der Ziel HDC (Z)
T  : Long - Kopiermodus
F  : Long - 0, oder Zeiger auf einen Bereich mit den Nummern der Tiles, die nur
kopiert werden sollen, alle anderen werden nicht kopiert
```

Ergebnis: 0

Eine sehr mächtige Funktion, die dem Programmierer jede Menge Arbeit abnimmt und sehr schnell arbeitet. Allerdings auch etwas kompliziert (15 Parameter), mal sehen, ob ihr meine Erkärung versteht...
Q ist der Zeiger auf ein Bild mit Tiles. Ein Tile ist ein rechteckiges Stück Grafik. Die komplette Landschaft wird also aus verschiedenen kleine Grafikstückchen zusammen gesetzt, in Spielen eine häufig praktizierte und speichersparende Technik.
Jedes Tile in Q muß gleich groß sein und alle Tiles müßen horizontal nebeneiander liegen. Erlaubt sind maximal 256 verschiedene Tiles, das sollte allemal ausreichen.
YT gibt an, ab welcher Y-Position in der Bitmap Q die Grafikplättchen zu finden sind.
AX und AY bestimmen letztendlich die Größe der zusammengesetzen Grafik , es wird angegeben, wieviele Tiles nach rechts und unten in die Zielbitmap Z kopiert werden sollen.
B und H geben Breite und Höhe eines einzelnen Tiles an, gemessen in Pixel.
M ist der Zeiger auf einen Bereich mit Mapdaten. In diesem Bereich steht jedes Byte für die Nummer eines Tiles in Q. 0 wäre also die Nummer des Tiles, das sich ganz links in der Bitmap befindet. Rechts daneben ist Tile 1, daneben Tile 2, dann Tile 3, usw.
BM gibt an, wieviele Tiles die Map M breit ist. Wenn ein Spielfeld also 128x64 groß wäre, dann müßte hier 128 stehen.
MX und MY geben den Offset innerhalb von M an, also die linke obere Position, ab der die Map als Grafik dargestellt werden soll.
Z gibt die Zielbitmap (HDC) an, in die die Tile-Grafiken kopiert werden. Das muß nicht zwangläufig %HDC sein.
OX und OY geben die Offsets in Z an, also die linke obere Ecke, ab der die Tile-Grafiken kopiert werden sollen.
T gibt den Kopiermodus an, alle Kopiermodi dürfen benutzt werden.
In F kann ein Bereich mit den Nummern der Tiles angegeben werden, die kopiert werden sollen. Alle anderen Tiles, die zwar in der Map vorhanden sind, aber im Bereich in F nicht aufgeführt wurden, werden nicht kopiert. Wird für F eine 0 angegeben, dann werden immer alle Tiles kopiert.
Wofür ist F denn überhaupt gut? Der Parameter F kann die Funktion SetTiles() ganz beträchtlich beschleunigen, z.B. wenn Figuren in ein Spielfeld gezeichnet werden sollen oder die Landschaft aus quasi 3-D Elementen zusammengesetzt wird. Angenommen wir haben eine Landschaft, die aus Gras, Wasser und hohen Bäumen besteht. Im ersten Aufruf von SetTiles() werden die unteren Teile der Grafikstücke kopiert, also Gras, Wasser und die Wurzeln der Bäume. Die Bäume sind aber viel höher als das Gras. Also werden in einem zweiten Durchlauf (noch mal SetTiles()) jetzt die oberen Teile (etwas nach oben verschoben) gesetzt. Bei den Bäumen ist das sinnvoll, beim Gras und Wasser aber nicht, weil bei zweiten Durchlauf ja eh nichts kopiert werden würde. Also bestimmen wir beim zweiten Durchlauf in F jetzt, das nur die Baum-Tiles kopiert werden sollen. So erreichen wir einen unglaublichen Geschwindigkeits-Gewinn.
Um das Ende des Bereichs in F zu kennzeichnen, muß der letzte Eintrag 1111 sein. Hier ein kleines Beispiel für die Generierung eines Bereichs in F:

```
text$="2,3,4,9,11,16,17,18,1111"
SetWords(welchetiles#,Addr(text$))
```

Ein Bereich in F muß immer mit Integern beschrieben (Word) werden.

Beispiele:

```
SetTiles(tileshdc&,25,24,16,32,25,map#,256,posx&,posy&,%HDC,0,64,0,0)
SetTiles(tileshdc&,0,24,16,32,25,map#,256,posx&,posy&,%HDC,0,39,-1,welchetiles#)
```

Den Anfang eines Spieles auf Tile-Basis findest ihr bei den ProSpeed Sourcecodes: **FA_Tiles.prf**.

CountFirstBytes (V,A,N,B)

Testet einen Teil eines Bereiches (oder den ganzen) auf ein bestimmtes Byte und meldet, wie oft das Byte anfangs in dem Bereich vorkommt, bis ein anderes Byte gefunden wird. Sinnvoll, z.B. um zu ermitteln, wieviele Leerzeichen anfangs in einem String vorkommen, bevor der reguläre Text erreicht ist.

```
V : Bereichsvariable
A : Long - Offsetadresse
N : Long - Anzahl zu durchsuchender Bytes
B : Long - Zu suchendes Byte
```

Ergebnis: Long - Position, an der ein anderes Byte als B vorkommt

Beispiel:

```
' Code läuft ab Profan 7, frühere Profanversionen müßen mit Bereichsvariablen
arbeiten...

string$="           Hallo"
x&=CountFirstBytes(Addr(string$),0,Len(string$),32)      'Chr$(32) = Leerzeichen
Print "Anzahl führender Leerzeichen: "+Str$(x&)
```

```
Anzahl führender Leerzeichen:12
```

CountBytes (V,A,N,B)

Testet einen Teil eines Bereiches (oder den ganzen) auf ein bestimmtes Byte und meldet, wie oft das Byte in dem Bereich vorkommt.

```
V : Bereichsvariable
A : Long - Offsetadresse
N : Long - Anzahl zu durchsuchender Bytes
B : Long - Zu suchendes Byte
```

Ergebnis: Long - Anzahl gefundener Bytes

Beispiel:
(Testet, wie oft das Byte 128 in einem 40000 Bytes großen Bereich vorkommt.

```
x&=CountBytes(bereich#,0,40000,128)
Print "Anzahl Bytes = "+Str$(x&)
```

```
Anzahl Bytes= 5867
```

CountWords (V,A,N,W)

Testet einen Teil eines Bereiches (oder den ganzen) auf ein bestimmtes Word und meldet, wie oft das Word in dem Bereich vorkommt.
Bitte beachten sie, das in einem Bereich, der z.B. mit 10000 dimensioniert wurde, nur 5000 Words vorkommen dürfen (1 Word = 2 Bytes).

```
V : Bereichsvariable
A : Long - Offsetadresse
N : Long - Anzahl zu durchsuchender Words
W : Long - Zu suchendes Word
```

Ergebnis: Long - Anzahl gefundener Words

Beispiel (Profan 7):
(Testet, wie oft das Integer / Word 4064 in einem 40000 Bytes großen Bereich vorkommt.

```
x&=CountBytes(bereich#,0,20000,4064)
Print "Anzahl Integer = "+Str$(x&)
```

AnzahlInteger= 644

CountLongs (V,A,N,L)

Testet einen Teil eines Bereiches (oder den ganzen) auf ein bestimmtes Longword (Long) und meldet, wie oft das Longword in dem Bereich vorkommt.
Bitte beachten sie, das in einem Bereich, der z.B. mit 10000 dimensioniert wurde, nur 2500 Longs vorkommen dürfen (1 Long = 4 Bytes).

```
V : Bereichsvariable
A : Long - Offsetadresse
N : Long - Anzahl zu durchsuchender Longwords
L : Long - Zu suchendes Longword
```

Ergebnis: Long - Anzahl gefundener Longwords

Beispiel (Profan 7):
(Testet, wie oft das LongInt 178064 in einem 40000 Bytes großen Bereich vorkommt.

```
x&=CountBytes(bereich#,0,10000,178064)
Print "Anzahl Integer = "+Str$(x&)
```

AnzahlInteger= 47

CountStrings (V,A,N,S,L)

Testet einen Teil eines Bereiches (oder den ganzen) auf den Inhalt eines Strings und meldet, wie oft der String in dem Bereich vorkommt.

```
V : Bereichsvariable
A : Long - Offsetadresse
N : Long - Anzahl zu durchsuchender Bytes (Länge des Bereichs, der durchsucht
werden soll)
S : String (oder Bereich), der den Suchstring enthält
L : Long - Länge des Suchstrings in Bytes
```

Ergebnis: Long - Anzahl gefundener Strings

Beispiel (Profan 7):
(Testet, wie oft der String "Hallo" in einem 80000 Bytes großen Bereich vorkommt.

```
text$="Hallo"
x&=CountStrings(bereich#,0,80000,addr(text$),Len(text$))
Print Chr$(34)+text$+Chr$(34)+" kam "+Str$(x&)+" x im Bereich vor."
```

"Hallo"kam 61 x im Bereichvor.

Version ()

Gibt einige Informationen zur ProSpeed.dll aus.

Ergebniss: Keins

AddWindowStyle (W,S,F)

Aktiviert ein (oder mehrere) Windows-Flags und Extended Window-Flags, und ändert somit das Aussehen eines Fensters.

```
W : Long - Handle eines Fensters oder (z.B. %HWnd) oder eines Controls (z.B.
Listbox)
S : Long - Wert(e) für Window-Style(s), bzw. Extended Window-Style(s)
F : Long - Flag, welcher Style benutzt wird: 0 = Window-Style / 1 = Extended
Window-Style
```

Ergebnis: 0

Hiermit kann das Aussehen eines Fensters verändert werden, das schon geöffnet ist. AddWindowStyle() fügt hierzu den bei S angegebenen Style hinzu. Dieser kann durchaus eine Kombination mehrerer Window-Styles sein. Allerdings dürfen die normalen Windows-Styles nicht mit den extended Window-Styles kombiniert werden! Wenn sie einen (oder mehrere) extended Window-Style(s) hinzufügen möchten, dann muß für F eine 1 angegeben werden, bei normalen Windows-Styles eine 0.

Beispiele:
(1. Fügt dem Profanfenster einen horizontalem und vertikalen Scroller hinzu)
(2. Fügt einer Listbox einen Fensterrahmen hinzu und macht das Fenster zu einem Toolwindow. Ja, geht wirklich !!!)

```
Window 0,0-800,600                  'Beispiel 1
AddWindowStyle(%hwnd,$300000,0)     '(beide Scroller hinzufügen)

listbox%=Create("Listbox",%hwnd,"",176,8,496,368)       'Beispiel 2
AddWindowStyle(listbox%,$C00000,0)                      '(Rahmen hinzufügen)
AddWindowStyle(listbox%,1,1)                                    '(Toolwindowstyle
hinzufügen)
SetText listbox%,"Dies' ist immer noch eine Listbox !"  '(Fenstertitel setzen)
```

SubWindowStyle (W,S,F)

Deaktiviert ein (oder mehrere) Windows-Flags und Extended Window-Flags, und ändert somit das Aussehen eines Fensters.

```
W : Long - Handle eines Fensters (z.B. %HWnd)
S : Long - Wert(e) für Window-Style(s), bzw. Extended Window-Style(s)
F : Long - Flag, welcher Style benutzt wird: 0 = Window-Style / 1 = Extended
```

```
Window-Style
```

Ergebnis: 0

Hiermit kann das Aussehen eines Fensters verändert werden, das schon geöffnet ist. SubWindowStyle() nimmt hierzu den bei S angegebenen Style weg. Dieser kann durchaus eine Kombination mehrerer Window-Styles sein. Allerdings dürfen die normalen Windows-Styles nicht mit den extended Window-Styles kombiniert werden! Wenn sie einen (oder mehrere) extended Window-Style(s) wegnehmen möchten, dann muß für F eine 1 angegeben werden, bei normalen Windows-Styles eine 0.

Beispiele:
(1. Nimmt den horizontalem und vertikalen Scroller aus dem AddWindowStyle-Beispiel wieder weg)
(2. Nimmt den Fensterrahmen den Toolwindow-Style aus dem AddWindowStyle-Beispiel wieder weg)

```
SubWindowStyle(%hwnd,$300000,0)     'Beispiel 1
```

```
SubWindowStyle(listbox%,$C00000,0) 'Beispiel 2
SubWindowStyle(listbox%,1,1)
```

GetSystemKey ()

Ermittelt die Taste, die gerade gedrückt wurde. Diese Funktion funktioniert systemweit und nicht nur im Profan-Fenster. Sogar die Maustasten werden mit abgefragt.

Ergebniss: ASCII-Code der gedrückten Taste. Wenn keine Taste gedrückt wurde, dann -1.

ChangeDisplay (X,Y)

Ändert die Bildschirmgrösse ohne sie zu speichern.
Gerade für Spiele benötigt man des öfteren eine Bildschirmauflösung von z.B. 1024 x 768. Bei den meisten Anwendern ist aber der Bildschirm auf 800 x 600 eingestellt. Mit ChangeDisplay() kann die Auflösung kurz mal eben umgestellt werden und anschließend wieder zurück.
Achtung, ab ProSpeed 2.1 kann ChangeDisplay() auch mit den Parametern 0,0 aufgerufen werden. Das bedeutet, der originale Windows-Bildschirm-Modus wird so wieder hergestellt, wie der Anwender ihn im System gespeichert hat. Somit entfällt jetzt das Speichern von oldx& und oldy& und die originale Bildfrequenz wird auch wieder hergestellt.
Neu ab Version 2.7 ist, das für jeden Modus versucht wird, die eingestellte Bildfrequenz zu erhalten. Und beim Zurückschalten des Bildschirms werden jetzt nicht mehr die Fenstergrössen anderer Programme verändert !

```
X : Long - Breite der neuen Bildschirmauflösung
Y : Long - Höhe der neuen Bildschirmauflösung
```

Ergebnis: Long - Fehlercode (0 =kein Fehler, ansonsten siehe MS_Win32.hlp, ChangeDisplaySettings())

Beispiel:

```
...
ChangeDisplay(1024,768)              'Bildschirm auf 1024 x 768 setzen
...
ChangeDisplay(0,0)                   'Bildschirm auf original Windowswerte
zurücksetzen
```

Compute (B,A,F,+,-,*,/,S)

Wendet einfache Rechenoperationen an allen oder einigen Long-Integer (optional auch Integer oder Bytes) in einem Bereich an.

```
B : Long - Bereichsvariable (Speicheradresse)
A : Long - Anzahl Bytes in B, die mit Compute() neu berechnet werden sollen
F : Long - Flag, welches Zahlensystem verwendet wird: 1 = Byte / 2 = Integer / 4
= LongInt
+ : Long - Dieser Wert wird zu jeder Zahl im Bereich addiert
- : Long - Dieser Wert wird zu jeder Zahl im Bereich subtrahiert
* : Long - Dieser Wert wird mit jeder Zahl im Bereich multipliziert
/ : Long - Jede Zahl im Bereich wird mit diesem Wert dividiert (Nachkomma-
Stellen werden abgeschnitten)
S : Long - Nach jeder Zahl so viele Bytes überspringen, normalerweise 0
```

Ergebnis: Long - 0

Mit Compute() können viele Zahlen gleichzeitig addiert, subtrahiert, muliplizierт und/oder dividiert werden.
Hierbei ist einstellbar (in F), ob die Zahlen im Bereich B Bytes, Integer oder LongInt's sein sollen. Eine 1 bedeutet, alle Zahlen werden als Bytes behandelt. Bei 2 werden alle Zahlen als Integer behandelt und bei 4 als LongInteger.
In S kann angegeben werden, ob jede Zahl im Bereich neu berechnet werden soll, oder ob nach jeder Zahl eine gewisse Anzahl Bytes uebersprungen werden sollen.
+, -, * und / können alle gleichzeitig angewendet werden, oder einzeln oder sonstwie kombiniert, die Reihenfolge, sollte mit allen vieren gerechnet werden, ist immer: multipizieren - dividieren - addieren - subtrahieren.
Wenn in F bestimmt wird, das Bytes oder Integer benutzt werden (1 oder 2), muß beachtet werden, das Profan Bytes und Integer in Bereichen immer als vorzeichenlos behandelt, also nur mit positiven Zahlen rechnet. Negative Zahlen erhalten so also falsche Werte. Bei Long-Integer ist das nicht so, diese werden immer vorzeichenbehaftet behandelt und liefern auch bei negativen Zahlen ein richtiges Ergebniss.
Bytes können nur innerhalb der Zahlen 0 - 255 operieren, Integer innerhalb der Zahlen 0 - 65535. Long-Integer können mit Zahlen von -2.147.483.648 bis 2.147.483.647 rechnen.

Beispiel 1:

```
text$="10,20,30,40"
SetBytes(bereich#,0,addr(text$))
Compute(bereich#,4,1,0,0,0,10,0)

'Ergebniss der ersten vier Byte in Bereich#: 1, 2, 3, 4 (alle Bytes / 10)
```

Beispiel 2:

```
text$="10,20,30,40"
SetWords(bereich#,0,addr(text$))
Compute(bereich#,8,2,0,0,10,0,0)

'Ergebniss der ersten vier Integer in Bereich#: 100, 200, 300, 400 (alle Integer
* 10)
```

Beispiel 3:

```
text$="10,20,30,40"
SetLongs(bereich#,0,addr(text$))
Compute(bereich#,16,4,11,22,0,0,0)

'Ergebniss der ersten vier LongInt in Bereich#: -1, 9, 19, 29 (alle LongInts +
```

```
11 - 22)
```

Beispiel 4:

```
text$="10,10,20,20,30,30,40,40"
Clear bereich#
SetLongs(bereich#,0,addr(text$))
Compute(bereich#,32,4,0,0,123,321,4)

'Ergebniss der ersten acht LongInt in Bereich#: 3, 10, 7, 20, 11, 30, 15, 40
'(jedes zweite LongInt * 123 / 321)
```

CopyMemory (Q,A1,Z,A2,B)

Kopiert den Inhalt eines Speicherblocks an eine andere Stelle irgendwo im Speicher.
Der Befehl arbeitet von links nach rechts. Er sollte nur verwendet werden, solange keine Überschneidung von Quell und Zielbereich stattfindet.

```
Q  : Long - Quelladresse (z.B. Bereichsvariable)
A1 : Long - Offset zur Quelladresse
Z  : Long - Zieladresse (z.B. Bereichsvariable)
A2 : Long - Offset zur Zieladresse
B  : Long - Anzahl Bytes, die kopiert werden sollen
```

Ergebnis: Long - 0

Beispiele:

```
CopyMemory (Tabelle1#,0,Tabelle2#,0,10000)
CopyMemory (Tabelle1#,4000,Tabelle1#,3000,1000)
```

Ausführdauer, 40000 Bytes:
Original-Profan = 4850 Millisekunden (kopieren mit WhileLoop-Schleife)
ProSpeed.dll = < 1 Millisekunden

CopyMemoryFromBack (Q,A1,Z,A2,B)

Funktionsbeschreibung siehe CopyMemory.
Beide Befehle haben die gleiche Wirkung und Syntax, arbeiten intern aber unterschiedlich. CopyMemoryFromBack arbeitet von rechts nach links. Bei dieser Methode können auch Überschneidungen von Quellbereich und Zielbereich richtig kopiert werden.
CopyMemory und CopyMemoryFromBack sind in ihrer Ausführung gleich schnell.
Zuerst ein Beispiel für CopyMemory:

```
CopyMemory (Tabelle#,0,Tabelle#,1,8)
```

8 Bytes sollen innerhalb eines Bereichs um 1 Byte nach rechts hin verschoben werden. Die Bytes werden von links nach rechts kopiert. CopyMemory wird hierbei aber verkehrte Werte schreiben.

```
          ----------------------»
Aus dieser Bytefolge 0->1->2->3->4->5->6->7->8  9  10  11 ...
wird                 0  0  0  0  0  0  0  0  0  9  10  11 ...
```

Byte 0 wird nach Byte 1 kopiert, dann wird Byte 1 (das jetzt den Wert von Byte 0 hat) nach Byte 2 kopiert,

Byte 2 nach Byte 3 usw.
Jetzt der gleiche Befehl als CopyMemoryFromBack:

```
CopyMemoryFromBack (Tabelle#,0,Tabelle,1,8)
```

Diesmal wird der Speicherbereich aber von rechts nach links kopiert.

```
          «-----------------------
Aus dieser Bytefolge 0->1->2->3->4->5->6->7->8  9  10  11 ...
wird                 0  0  1  2  3  4  5  6  7  9  10  11 ...
```

Byte 7 wird nach Byte 8 kopiert, dann wird Byte 6 nach Byte 7, Byte 5 nach Byte 6 usw.
CopyMemoryFromBack arbeitet also korrekt.

Anders herum würde bei folgendem Befehl der CopyMemory Befehl richtig arbeiten und CopyMemoryFromBack würde falsche Werte in den Speicher schreiben:

```
CopyMemory (Tabelle#,1,Tabelle,0,8)
```

8 Bytes sollen diesmal innerhalb eines Bereichs um 1 Byte nach links hin verschoben werden.

ChangeBytesToWords (Q,A1,Z,A2,N)

Verwandelt einige oder alle Bytes in einem Bereich nach Word um.
Eine Überschneidung von Quell und Zielbereich ist nicht ratsam, verwenden sie zwei verschiedene Bereiche, oder verschiedene Abschnitte innerhalb eines Bereichs. Noch zu bedenken ist, das der Zielbereich doppelt so groß sein muß wie der Quellbereich, da ein Word ja zwei Bytes beinhaltet.

```
Q  : Long - Quelladresse (z.B. Bereichsvariable)
A1 : Long - Offset zur Quelladresse
Z  : Long - Zieladresse (z.B. Bereichsvariable)
A2 : Long - Offset zur Zieladresse
N  : Long - Anzahl Bytes, die zu Words verwandelt werden sollen
```

Ergebniss: Long - 0

Beispiel:

```
ChangeBytesToWords (Bereich#,0,Bereich#,20000,10000)
```

ergibt:

```
Adresse: 0      1      2      3      4      5     ...   20000  20002  20004
 20006  20008  ...
Wert:    111    111    111    111    111    111   ...   111    111    111    111
   111
```

Ausführdauer, 40000 Bytes:
Original-Profan = 7010 Millisekunden
ProSpeed.dll = 1 Millisekunde

ChangeWordsToLong (Q,A1,Z,A2,N)

Verwandelt einige oder alle Words in einem Bereich nach Long um.

Eine Überschneidung von Quell und Zielbereich ist nicht ratsam, verwenden sie zwei verschiedene Bereiche, oder verschiedene Abschnitte innerhalb eines Bereichs. Noch zu bedenken ist, das der Zielbereich viermal so groß sein muß wie der Quellbereich, da ein Long ja zwei Words (4 Bytes) beinhaltet.

```
Q  : Long - Quelladresse (z.B. Bereichsvariable)
A1 : Long - Offset zur Quelladresse
Z  : Long - Zieladresse (z.B. Bereichsvariable)
A2 : Long - Offset zur Zieladresse
N  : Long - Anzahl Words, die zu Longs verwandelt werden sollen
```

Ergebniss: Long - 0

Beispiel:

```
ChangeWordsToLongs (Bereich#,0,Bereich#,20000,5000)
```

ergibt:

```
Adresse: 0      1      2      3      4      5     ...   20000  20004  20008
 20012  20016  ...
Wert:    11123  11123  11123  11123  11123  11123 ...   11123  11123  11123
 11123  11123
```

Ausführdauer, 40000 Words:
Original-Profan = 10070 Millisekunden
ProSpeed.dll = 1 Millisekunde

ChangeLongSequence (B,A)

Kehrt die Reihenfolge der Long-Integer in Bereich B um.

```
B : Long - Bereichsvariable, Tabelle mit LongInt-Werten
A : Long - Anzahl Bytes, die bei der Umkehr betroffen sind (glatt teilbar durch
4 natürlich !)
```

Ergebniss: Long - 0

Beispiel, um die Reihenfolge der Long-Werte in bereich# zu ändern. Aus: 8, 167800, 56000, 45 würde 45, 56000, 167800, 8.

```
ChangeLongSequence(bereich#,16)    '16 = 4 Longs a 4 Byte, 4 x 4 = 16
```

ClearWith (V,A,B,N)

Löscht einen Teil eines Bereiches (oder den ganzen) mit einem bestimmten Wert (Byte-Format).
In Anlehnung an den Profan-Befehl *Clear* wurde diese Funktion implementiert, der aber Bereiche nicht nur mit Nullen löschen kann, sondern mit allen Werten zwischen 0 und 255.

```
V : Long - Bereichsvariable
A : Long - Offset
B : Long - Byte, mit dem gelöscht werden soll (0-255)
N : Long - Anzahl zu löschender Bytes
```

Ergebniss: Long - 0

Beispiel (beschreibt einen 200000 Byte großen Bereich mit dem Bytewert 128):

```
ClearWith(bereich#,0,128,200000)
```

CompareBytes (B1,B2,A,T)

Vergleicht A Bytes der beiden Speicherbereiche B1 und B2 auf Gleichheit bzw. Unstimmigkeiten.

```
B1 : Long - Bereichsvariable 1
B2 : Long - Bereichsvariable 2
A  : Long - Anzahl Bytes, die überprüft werden sollen
T  : Long - Toleranz für jeden Byte-Vergleich
```

Ergebniss: Long - 0 bei kompletter Übereinstimmung, ansonsten Anzahl verschiedener Bytes.

Eine oft benötigte Funktion, besonders geeignet um Texte oder Bitmaps miteinander zu vergleichen, und wie immer sehr schnell.
T wird im Normalfall Null sein. Sie können aber auch eine Toleranz mit angeben. Wenn die beiden Bytes aus den beiden Bereichen verglichen werden, und der Unterschied zwischen beiden ist nicht größer ist wie der Toleranzwert, dann fällt der Vergleich positiv aus, beide Byte-Werte werden also als gleich betrachtet. Bei einer Tolerant von 10 z.B. würden die beiden Zahlen 201 und 208 als noch gleich bewertet.

Beispiele:
(1. Vergleicht 80000 Bytes Daten von zwei Bereichen, ohne Toleranz)
(2. Vergleicht zwei Bilder (bmp und jpg, gleiches Motiv) miteinander, 4 x mit unterschiedlichen Toleranzen (0, 10, 50, 100)).

```
x&=CompareBytes(bereich1#,bereich2#,80000,0)    'Beispiel 1
Print "Anzahl verschiedener Bytes: "+Str$(x&)
```

```
text$="Picture                                  'Beispiel 2
pic1&=LoadFileImage(addr(text$))

text$="Picture.jpg"
pic2&=LoadFileImage(addr(text$))

array1&=InitFX(pic1&)
array2&=InitFX(pic2&)

bytes&=Long(array1&,20) ;Größe der Bilder in Bytes (beide gleich groß)
x&=Long(array1&,40) ;Zeiger auf Byte-Array von Bild 1
y&=Long(array2&,40) ;Zeiger auf Byte-Array von Bild 2

z&=CompareBytes(x&,y&,bytes&,0)
Print "Anzahl verschiedener Farbpixel (von insgesamt "+Str$(bytes&)+"): "+Str$
(z&)
z&=CompareBytes(x&,y&,bytes&,10)
Print "Anzahl verschiedener Farbpixel (von insgesamt "+Str$(bytes&)+"): "+Str$
(z&)
z&=CompareBytes(x&,y&,bytes&,50)
Print "Anzahl verschiedener Farbpixel (von insgesamt "+Str$(bytes&)+"): "+Str$
(z&)
z&=CompareBytes(x&,y&,bytes&,100)
Print "Anzahl verschiedener Farbpixel (von insgesamt "+Str$(bytes&)+"): "+Str$
(z&)
```

```
FreeFX(array1&)
FreeFX(array2&)

FreeImage(pic1&)
FreeImage(pic2&)
```

Crypt (B,A,S,L)

Verschlüsselt Daten in einem Bereich 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
```

Ergebniss: Long - 0

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

```
name$="Datenfile.dat"
x&=FileSize(name$)
ReadFileFast(addr(name$),bereich#,0,x&)            'Datei/Daten in
Bereichsvariable einladen

passwort$="CryptMe"                                'Passwort ist "CryptMe"
Crypt(bereich#,x&,addr(passwort$),len(passwort$))  'Daten aus dem Bereich
verschlüsseln

name$="Datenfile_crypted.dat"                      'Hier könnte die
verschlüsselte Datei wieder
WriteFileFast(addr(name$),bereich#,0,x&)           'gespeichert werden...

Crypt(bereich#,x&,addr(passwort$),len(passwort$))  '...oder wieder in den
originalen Zustand gebracht
                                                   'werden
```

GenerateChecksum (B,A,S)

Generiert eine Checksumme aus A Bytes des Speicherbereichs B. Als Startwert kann für S ein anderer Wert als Null angegeben werden.

```
B : Long - Bereichsvariable
A : Long - Anzahl Bytes, die für die Checksumme verwendet werden
S : Long - Startwert der Checksumme
```

Ergebniss: Long - Checksumme.

Checksummen können sehr vielfältig verwendet werden. Sie werden sehr einfach berechnet. Jedes Byte des angegebenen Speicherbereichs wird zu der Variablen (S) addiert. So entsteht ein recht individuelles Erkennungszeichen für einen Speicher.

Beispiel (Generiert eine Checksumme aus 80000 Bytes von bereich#):

```
GenerateChecksum(bereich#,80000,0)
```

FindBytes (B,O,A,S,L)

Eine Kopie der Profan-Funktion MemPos, hat aber zwei Paramerter mehr, läuft dafür aber ab Profan 5 und ist über 100 mal schneller !

```
B : Long - Bereichsvariable
O : Long - Offset zum Bereich
A : Long - Anzahl zu durchsuchender Bytes im Bereich
S : Long - Zeiger auf einen Suchstring (ab Profan 7) oder zweite
Bereichsvariable, z.B. Addr(string$)
L : Long - Länge des Strings oder zweiten Bereichs, z.B. Len(string$)
```

Ergebniss: Long - Fundstelle oder -1, wenn der String nicht gefunden wird.
Mit dieser Funktion kann eine beliebige Zeichenfolge in einem Bereich gesucht werden.

Beispiel:

```
string bereich#,700000="Frank"  'Bereich mit den Bytes "Frank" beschreiben, hier
mit 800000 dimensioniert
text$="Frank"                   'Den Such-String "Frank" erstellen
x&=FindBytes(bereich#,0,800000,Addr(text$),Len(text$))     'x& zeigt jetzt
700000 an
```

PixelToMemory (H,B,S)

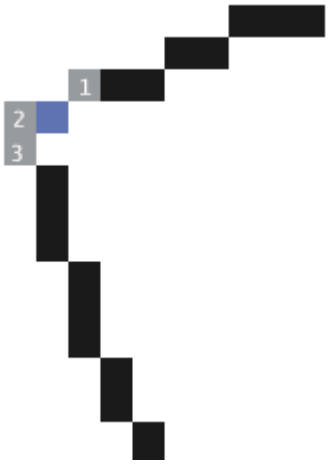
Die Grafik im Byte-Array H wird pixelweise abgetastet und die ermittelten X- und Y-Koordinaten werden im Bereich B als Tabelle gespeichert. Sehr nützlich zusammen mit MoveSpriteWithTable()

```
H : Long - Handle eines mit CreateFX, PngToFX() oder InitFX erzeugten Bytes-
Arrays
B : Long - Bereich, im dem die Koordinaten-Tabelle generiert wird (ausreichend
dimensionieren!)
S : Long - Schrittweite in Pixeln, mit der die Grafik abgetastet wird, 1 bis ...
(nie Null!)
```

Ergebniss: Long - Anzahl Bytes (!), die im Bereich B eingetragen wurden.

Ab Version 2.1 kann ProSpeed Sprites in jeder möglichen Flugbahn bewegen. Hierzu bedient es sich einer speziellen Tabelle mit X-und Y-Koordinaten.
Mit PixelToMemory() kann von fast jedem Grafikobjekt so eine Tabelle generiert werden, sofern die Grafik folgende Kriterien erfüllt.

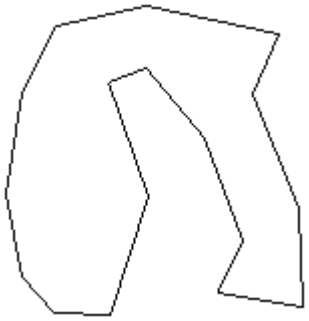
```
1) die Pixel der Grafik müßen schwarz sein (also den Hintergrund am besten weiß
lassen)
2) jeder Pixel darf höchstens zwei Nachbar-Pixel haben, die entweder oben,
unten, links, rechts oder in    einer der vier diagonalen Ecken sitzen dürfen.
Also sind auch keine überschneidenen Linien und keine    Linien, die dicker als
einen Pixel sind, erlaubt.
3) die Grafiklinie darf nicht unterbrochen sein
4) die Linie darf die Bitmap nicht verlassen
```



Links die Grafik zeigt den vergrößerten Ausschnitt einer fehlerhaften Linie, wobei der Fehler beim blauen Pixel liegt (einige Pixel sind hier nur farbig dargestellt und nummeriert worden, um den Fehler zu demonstrieren).
Der blaue Pixel besitzt drei Nachbarpixel (siehe Kriterium 2), was beim Berechnen der Koordinatentabelle zu Fehlern führt.
Wenn sie also PixelToMemory() benutzen, dann müssen sie alle Kriterien beachten und notfalls die Grafik von Hand nacharbeiten. Links in der Abbildung würde es sich z.B. anbieten, Pixel Nr. 2 zu löschen...

Die Tabelle B, die generiert wird, besteht anschließend aus Integer-Zahlen (Words), immer ein Word für die X-Koordinate, dann eine Word für die Y-Koordinate, dann X, Y, X, Y,....usw. Jeweils ein X und ein Y-Wert ergeben einen Punkt auf dem Bildschirm.
Hier mal ein Beispiel:

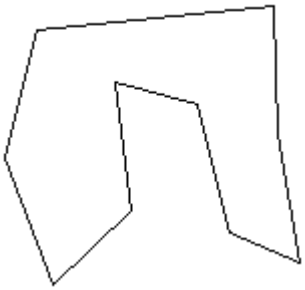
Dieses Bild (160x160, schwarz-weiss,gespeichert als .gif) wurde mit einem Paintprogramm via Polygon-Modus gemalt (bitte die Kiterien beachten). Jetzt wandeln wir das Bild in Koordinatenpaare um, via:



PixelToMemory(array&,bereich#,1)

Wenn wir mittels der generierten Tabellenwerte jetzt wieder Linien malen würden, würde das Originalbild wieder genauso aussehen (in diesem Fall 702 Koordinatenpaare).

Jetzt wandeln wir mal das gleiche Bild in Koordinatenpaare um, diesmal mittels:



PixelToMemory(array&,bereich#,64)

Jetzt würden wir dieses Bild malen, wenn wir unsere Koordinaten wieder als Linien malen (in diesem Fall 10 Koordinatenpaare):

Die Schrittweite S gibt also an, vieviele Koordinaten im Vergleich zu den ausgelesenen Bild-Pixeln in die Tabelle in B eingetragen werden. Ein Wert von 1 gibt an, das die Position jedes Pixels als Tabellenwert verwendet wird, ein Wert von 30 z.B. gibt an, das nur jeder 30ste Pixel eingetragen wird. Je höher der Wert in S ist, desto genauer wird die Linie im Bild H als Tabelle gespeichert, und desto weniger Koordinatenpaare werden berechnet.

Hier ist übrigens der Profan 7-Code, um eine Koordinatenpaar-Tabelle wieder in Linien umzusetzen:

```
text$="Testbild.gif"                    'Bild laden
pic&=LoadFileImage(addr(text$))          '
array&=InitFX(pic&)                  'und in eine Byte-Array kopieren#

a&=PixelToMemory(array&,bereich#,64)

x&=Word(bereich#,0)
y&=Word(bereich#,2)
MoveTo x&,y&

z&=0
Whileloop a&/4
x&=Word(bereich#,z&)
y&=Word(bereich#,z&+2)
LineTo x&,y&
z&=z&+4
Wend

x&=Word(bereich#,0)
y&=Word(bereich#,2)
LineTo x&,y&

FreeFX(array&)
FreeImage(pic&)
```

Random (B,L,A,M)

Schreibt eine beliebige Anzahl generierter Zufallswerte in einen Bereich.

```
B : Long - Bereich, der die Zufallszahlen aufnehmen soll.
L : Long - Anzahl Longs, die in den Bereich geschrieben werden sollen (1 Long =
4 Byte)
A : Long - Wert, der zu jeder Zufallszahl addiert werden soll, oder 0
M : Long - Maximumwert der zu generierenden Zufallszahlen
```

Ergebniss: Long - 0

Alle Zufallszahlen, die generiert werden, werden als LongInteger in den Speicher (Bereich) geschrieben. Der Bereich muß vorher natürlich ausreichend dimensioniert werden.
Es werden Zahlen erzeugt, die zwischen 0 und M-1 liegen, anschließend wird noch A hinzu addiert (A darf auch negativ sein).

Beispiele:

```
'*** Erzeugt in Bereich# 100 Zufallszahlen zwischen 1000 und 1199)
Random (bereich#,100,1000,200)

'*** Erzeugt in Bereich# 4000 Zufallszahlen zwischen -1000 und 0)
Random (bereich#,4000,-1000,1000)

'*** Erzeugt in Bereich# 30 Zufallszahlen zwischen 0 und 1)
Random (bereich#,30,0,2)
```

ReplaceTabs (Q,Z,A,L)

Wandelt alle Tab's (Chr$(9)) im Bereich Q um in eine bestimmte Anzahl Leerzeichen.

```
Q : Long - Quellbereich
Z : Long - Zielbereich
A : Long - Anzahl zu durchsuchender Bytes im Quellbereich
L : Long - Anzahl einzusetzender Leerzeichen pro Tab
```

Ergebniss: Long - Anzahl Leerzeichen, die insgesamt eingesetzt wurden.

Ein logischer Einsatz für ReplaceTabs() wäre z.B. der Text eines Editors.
Der Quellbereich ist dann der originale Text, im Zielbereich landet dann der neue Text mit den Leerzeichen.
A wäre die Länge des alten Texts. Bei L wird bestimmt, durch wieviele Leerzeichen ein Tab ausgetauscht wird.
Um zu berechnen, wie groß überhaupt der Zielbereich Z dimensioniert werden muß, bietet sich CountBytes() an.

Beispiel:

```
leerzeichen&=8                                  '8 Leerzeichen pro Tab
length&=100000                                  'Grösse des Texts in bereich#
x&=CountBytes(bereich#,0,length&,9)             'Tab's zählen, Chr$(9)
newlenght&=lenght&+(x&*(leerzeichen&-1))        'Neue Länge=
                    'Alte Länge + (Gezählte Tab's * (Anzahl Leerzeichen -1))
Dim newbereich#,newlenght&+1
ReplaceTabs(bereich#,newbereich#,lenght&,leerzeichen&)

'der neue Text steht jetzt in Bereich newbereich# und hat die Länge newlenght&.
```

Replace (Q,A,Z,S1,L1,S2,L2)

Alle Vorkommen des Strings S1 werden in einem Text oder Datenbereich durch S2 ersetzt. Funktioniert so wie Profan's "Translate$", kann aber auf beliebig Große Strings bzw. Bereiche angewendet werden. Replace() arbeitet noch dazu unglaublich schnell !

```
Q  : Long - Quellbereich
A  : Long - Größe des Datenbereichs in Q (Anzahl Bytes)
Z  : Long - Zielbereich
S1 : String mit dem Suchtext
L1 : Long - Länge des Suchtext-Strings in Bytes
S2 : String mit dem Ersetzentext
L2 : Long - Länge des Ersetzentext-Strings in Bytes
```

Ergebniss: Long - Größe des neu generierten Texts im Datenbereichs Z (Anzahl Bytes).

Replace() arbeitet CaseSensitive, Groß- und Kleinbuchstaben werden also gesondert behandelt.
Hiermit kann jetzt jeder eine Suchen / Ersetzen - Funktion für seine Texte oder Daten programmieren.
Was wir brauchen, sind zwei Datenbereiche, einen mit dem Quelltext Q und einen, in dem später der Zieltext generiert wird (Z).
Zunächst müssen wir wissen, wie groß der Zieltext überhaupt werden wird. Dabei hilft uns CountStrings().
Das folgende Beispiel zeigt, wie Teile eines großen Textes ausgetauscht werden können.

Beispiel (Komplettbeispiel für ab Profan 7):

```
'In bereich# wird eine Textdatei geladen. In diesem Text sollen alle Punkte (die
Satzzeichen sind gemeint) 'als Klartext "(Punkt)" umgewandelt werden. Der
Zieltext wird dadurch natürlich größer und anschließend als 'Datei gespeichert.

Declare
neu&,bereich#,bereich2#,length&,length2&,l1&,l2&,text1$,text2$,dateiname$,x&

$I Prospeed_Funktionen.inc

neu&=usedll("ProSpeed.dll")
SetTrueColor 1
Cls

'*** Beispiel: Textdatei Datei laden
'
dateiname$="Test.txt"
length&=FileSize(dateiname$)
dim bereich#,length&+1
x&=ReadFileFast(addr(dateiname$),bereich#,0,length&)
Print Str$(x&)+" Bytes gelesen."

text1$="."       'Suchtext, ersetzt hier "." durch
text2$="(Punkt)" 'den Ersetzentext "(Punkt)."
l1&=@Len(text1$) 'Länge des Suchtext
l2&=@Len(text2$) 'Länge des Ersetzentext

x&=CountStrings(bereich#,0,length&,addr(text1$),l1&) 'Zählen, wie oft der
Suchtext im Text vorkommt
length2&=length&-(x&*l1&)+(x&*l2&)                  'Datenlänge des neuen
Bereichs ermitteln

Dim bereich2#,length2&+1
x&=Replace(bereich#,length&,bereich2#,addr(text1$),l1&,addr(text2$),l2&) 'Text
umwandeln

Dispose bereich#                            'Die neu erstellten Daten befinden
sich jetzt
Dim bereich#,length2&+1                     'in bereich2#. In der Regel wird
unser Haupt-Programm
CopyMemory(bereich2#,0,bereich#,0,length2&) 'seine Daten aber immer in bereich#
bearbeiteten.
Dispose bereich2#                           'Also sollten wir die neu erstellten
Daten auch wieder
                  'in einen Bereich namens bereich# kopieren.
                  'bereich2# benötigen wir dann nicht mehr.
dateiname$="Test2.txt"
x&=WriteFileFast(addr(dateiname$),bereich#,0,length2&)
Print Str$(x&)+" Bytes geschrieben."

WaitInput
freedll neu&
End
```

FindJoystick (N)

Prüft nach, ob ein Joystick (1 oder 2) vorhanden ist.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - 0 = Joystick vorhanden / -1 = Joystick nicht vorhanden oder Fehler.

Wird 0 zurückgegeben, so ist dieser Joystick verfügbar. Bei Rückgabe von -1 kann dieser Joystick nicht benutzt werden. Entweder ist er nicht vorhanden, oder nicht (richtig) installiert.

```
x&=FindJoystick(1)
```

CountJoystickButtons (N)

Ermittelt die Anzahl Knöpfe, über die der angeschlossene Joystick N verfügt.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Anzahl der verfügbaren Joystickknöpfe, bei Fehler 0.

```
x&=CountJoystickButtons(1)
```

JoystickX (N)

Ermittelt die Links/Rechts Steuerung eines angeschlossenen Joysticks.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der Links/Rechts Steuerung eines Joysticks.

```
-128 = Joystick wird gerade ganz nach links gedrückt
   0 = Joystick wird in der Links/Rechts Achse nicht bewegt
 127 = Joystick wird ganz nach rechts gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Beim Wert 48 z.B. wird der Joystick nur ein bischen nach rechts gedrückt.
Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 112 oder kleiner als -112 sollten als ein volle Joystickbewegung gewertet werden.

Ab ProSpeed 2.0 braucht nicht mehr %hwnd angegeben werden. Ein Profanfenster ist für eine Joystickabfrage jetzt auch nicht mehr nötig. Vielmehr können sie nun mit 1 oder 2 wählen, welcher von zwei Joysticks angesprochen werden soll. Auch die Fehlerbehandlung hat sich geändert. Alte Sourcecodes müssen also ggf. angepaßt werden.

JoystickY (N)

Ermittelt die Oben/Unten Steuerung eines angeschlossenen Joysticks.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der Oben/Unten Steuerung eines Joysticks.

```
-128 = Joystick wird gerade ganz nach oben gedrückt
   0 = Joystick wird in der Links/Rechts Achse nicht bewegt
 127 = Joystick wird ganz nach unten gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Beim Wert 48 z.B. wird der Joystick nur ein bischen nach unten gedrückt.
Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 104 oder kleiner als -104 sollten als ein volle Joystickbewegung gewertet werden.

Ab ProSpeed 2.0 braucht nicht mehr %hwnd angegeben werden. Ein Profanfenster ist für eine Joystickabfrage jetzt auch nicht mehr nötig. Vielmehr können sie nun mit 1 oder 2 wählen, welcher von zwei Joysticks angesprochen werden soll. Auch die Fehlerbehandlung hat sich geändert. Alte Sourcecodes müssen also ggf. angepaßt werden.

JoystickZ (N)

Ermittelt die Z Achse eines angeschlossenen Joysticks.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der Z-Achse eines Joysticks.

```
-128 = Die Z-Achse des Joysticks wird gerade ganz zu einer Seite gedrückt
   0 = Joystick wird in der Z-Achse nicht bewegt
 127 = Die Z-Achse des Joysticks wird gerade ganz zur anderen Seite gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 104 oder kleiner als -104 sollten als ein volle Joystickbewegung gewertet werden.

JoystickR (N)

Ermittelt die R Achse eines angeschlossenen Joysticks (Ruder und Pedalen).

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der R-Achse eines Joysticks.

```
-128 = Die R-Achse des Joysticks wird gerade ganz zu einer Seite gedrückt
   0 = Joystick wird in der R-Achse nicht bewegt
 127 = Die R-Achse des Joysticks wird gerade ganz zur anderen Seite gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 104 oder kleiner als -104 sollten als ein volle Joystickbewegung gewertet werden.

JoystickU (N)

Ermittelt die U Achse eines angeschlossenen Joysticks (fünfte Achse).

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der U-Achse eines Joysticks.

```
-128 = Die U-Achse des Joysticks wird gerade ganz zu einer Seite gedrückt
   0 = Joystick wird in der U-Achse nicht bewegt
 127 = Die U-Achse des Joysticks wird gerade ganz zur anderen Seite gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 104 oder kleiner als -104 sollten als ein volle Joystickbewegung gewertet werden.

JoystickV (N)

Ermittelt die V Achse eines angeschlossenen Joysticks (sechste Achse).

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Aktueller Stand der V-Achse eines Joysticks.

```
-128 = Die V-Achse des Joysticks wird gerade ganz zu einer Seite gedrückt
   0 = Joystick wird in der V-Achse nicht bewegt
 127 = Die V-Achse des Joysticks wird gerade ganz zur anderen Seite gedrückt

 256 = Fehler (der Joystick oder diese Joystickachse existiert nicht)
```

Die Genauigkeit der System-Joystickabfrage ist nicht sehr weit her. Alle Werte zwischen -24 und +24 sollten als ein nichtbewegen der Joystickachse gewertet werden. Alle Werte von größer als 104 oder kleiner als -104 sollten als ein volle Joystickbewegung gewertet werden.

JoystickButton (N)

Ermittelt, welcher von bis zu 32 Knöpfen eines Joysticks gerade gedrückt wird.

```
N : Long - Nummer des Joysticks, 1 oder 2.
```

Ergebniss: Long - Welcher Knopf wird gerade gedrückt.

```
0  = kein Knopf (oder Fehler)
1  = Knopf A (Bit 0)
2  = Knopf B (Bit 1)
4  = Knopf C (Bit 2)
8  = Knopf D (Bit 3)
16 = Knopf E (Bit 4)
usw...
```

Bis zu 32 (!) Joystickknöpfe können hiermit jeweils für Joystick 1 und 2 abgefragt werden.
Die verschiedenen Werte können auch kombiniert sein. Ein zurückgegebener Wert von z.B. 9 bedeutet, das

Knopf A und Knopf D beide
augenblicklich gedrückt werden.
Zur Auswertung mit Profan bietet sich die Funktion @TestBit(N1,N2) an.

Ab ProSpeed 2.0 braucht nicht mehr %hwnd angegeben werden. Ein Profanfenster ist für eine Joystickabfrage jetzt auch nicht mehr nötig. Vielmehr können sie nun mit 1 oder 2 wählen, welcher von zwei Joysticks angesprochen werden soll. Auch die Fehlerbehandlung hat sich geändert. Alte Sourcecodes müssen also ggf. angepaßt werden.

```
x&=JoystickButton(1)

If TestBit(x&,0)      'Knopf A gedrückt ?
...
ElseIf TestBit(x&,1)  'Knopf B gedrückt ?
...
ElseIf TestBit(x&,2)  'Knopf C gedrückt ?
...
EndIf
```

SetMouseXY (X,Y)

Setzt den Mauszeiger an die Position X und Y.

```
X : Long - Neue X-Position des Mauszeigers
Y : Long - Neue Y-Position des Mauszeigers
```

Ergebniss: Long - TRUE oder bei Fehler 0

Diese Funktion bietet sich für eine bestimmte Art der Maussteuerung an, wie sie z.B. in 3-D Spielen oder Scroll-Spielen verwendet wird. Dazu setzten sie den Mauszeiger auf die Mitte ihres Fensters. Der Zeiger darf dazu nicht sichtbar sein (`ShowCursor 0`).
Bevor sie jetzt immer wiederkehrend den Mauszeiger auf die Mitte ihres Fensters setzen, fragen sie die X und Y Position des Mauszeigers ab (`%mausx, %mausy`) und können so erfahren, wie weit der Anwender den Mauszeiger in der Zwischenzeit bewegt hat. Diese Werte können sie addieren und haben so eine sehr sensible Maussteuerung.

```
SetMouseXY(100,200)
```

SetMouseRect (X1,Y1,X2,Y2)

Begrenzt die Verschiebung des Mauszeigers auf die angegebenen Koordinaten.

```
X1 : Long - Neue X-Position des Mauszeigers
Y1 : Long - Neue Y-Position des Mauszeigers
X2 : Long - Neue X-Position des Mauszeigers
Y2 : Long - Neue Y-Position des Mauszeigers
```

Ergebniss: Long - TRUE oder bei Fehler 0

Diese Funktion legt fest, innerhalb welchen Bereiches des Bildschirms die Maus bewegt werden darf. Bereiche außerhalb dieses Rechtecks kann der Anwender mit der Maus nicht erreichen. Eignet sich hervoragend für Spiele, um den Mauszeiger im Menü-Bereich zu halten und nicht z.B. auf der Spielfläche.
Zum Ende des Programmes sollten die Ausgangswerte aber wieder hergestellt werden durch:

```
SetMouseRect(0,0,%maxx,%maxy)
```

MouseButton ()

Ermittelt, welche Taste der Maus gerade gedrückt wird.

Ergebniss: Long - Welche Taste wird gerade gedrückt.

```
0 = keine Taste (oder Fehler)
1 = linke Taste (Bit 0)
2 = rechte Taste (Bit 1)
4 = mittlere Taste (Bit 2)
```

Der Ergebniss-Wert kann auch kombiniert sein. Ein zurückgegebener Wert von z.B. 5 bedeutet, das die linke Maustaste und die mittlere Maustaste beide augenblicklich gedrückt werden.
Zur Auswertung mit Profan bietet sich die Funktion `@TestBit(N1,N2)` an.

Diese Funktion hat einige Vorteile gegenüber den Profan-Variablen/Funktionen. Mit MouseButton() kann endlich auch die mittlere Maustaste abgefragt werden, sofern vorhanden. Außerdem können alle Maustasten gleichzeitig abgefragt werden, und der Status geht hier nicht mehr durch `WaitInput` oder `WaitMouse` verloren.

```
x&=MouseButton()

If TestBit(x&,0)      'Linke Maustaste gedrückt ?
...
ElseIf TestBit(x&,1)  'Rechte Maustaste gedrückt ?
...
ElseIf TestBit(x&,2)  'Mittlere Maustaste gedrückt ?
...
EndIf
```

SetBytes (V,A,S)

Schreibt eine unbegrenzte Zahl an beliebigen Bytes in einen Bereich (mit Dummy-Funktion).

```
V : Bereichsvariable
A : Long - Offsetadresse
S : String mit Eingabe-Instruktionen
```

Ergebnis: Long - Anzahl Bytes, die bearbeitet wurden.

Mit diesem Befehl ist es sehr elegant möglich, einen Bereich mit beliebigen Werten zu füllen. Ein Beispiel wäre:

```
text$="4,6,8,10,12,14,16,18,*,22,24,26,*,*,32,34"
SetBytes(Bereich#,0,addr(text$))
```

Die erste Zeile generiert die Eingabe-Instruktionen. Die Werte innerhalb der Anführungszeichen sind die eigentlichen Daten, die in den Bereich geschrieben werden sollen. Sie können hierbei soviele Werte auf einmal schreiben, wie sie wollen.

```
"4,6,8,10,12,14,16,18,*,22,24,26,*,*,32,34"
```

Der Stern (*) ist ein Dummy-Wert, an dieser Stelle werden die ursprünglichen Werte des Bereichs nicht überschrieben.
Die oben stehenden Befehle würden in einen leeren Bereich (nur Nullen) folgendes hinein schreiben.

```
Wert:    4   6   8  10  12  14  16  18  0   22  24  26  0   0   32  34
```

Drei Null-Werte wurden also nicht überschrieben.

Der kleine Umweg der Daten-Eingabe über einen String / zweiten Bereich war leider nicht zu umgehen, weil Profan höchstens 15 Parameter an DLL's senden kann, und die Anzahl darf nicht einmal variabel sein.

SetWords (V,A,S)

Schreibt eine unbegrenzte Zahl an beliebigen Words in einen Bereich (mit Dummy-Funktion).

```
V : Bereichsvariable
A : Long - Offsetadresse
S : String mit Eingabe-Instruktionen
```

Ergebnis: Long - Anzahl Words, die bearbeitet wurden.

Beispiel:

```
text$="4,6,8,10,12,14,16,18,*,22,24,26,*,*,32,34"
SetWords(Bereich#,0,addr(text$)
```

Mit diesem Befehl ist es sehr elegant möglich, einen Bereich mit beliebigen Word-Werten zu füllen.
Eine detailierte Beschreibung finden sie unter SetBytes, nur das es hier um Words geht.

SetLongs (V,A,S)

Schreibt eine unbegrenzte Zahl an beliebigen Longs in einen Bereich (mit Dummy-Funktion).

```
V : Bereichsvariable
A : Long - Offsetadresse
S : String mit Eingabe-Instruktionen
```

Ergebnis: Long - Anzahl Longs, die bearbeitet wurden.

Beispiel:

```
text$="4,6,8,10,12,14,16,18,*,22,24,26,*,*,32,34"
SetLongs(Bereich#,0,addr(text$)
```

Mit diesem Befehl ist es sehr elegant möglich, einen Bereich mit beliebigen Long-Werten zu füllen.
Eine detailierte Beschreibung finden sie unter SetBytes, nur das es hier um Longs geht.

StartWatch ()

Startet eine Stoppuhr, deren Stand mit ReadWatch() gelesen werden kann.
Diese Uhr ist besonders ideal für Messungen während der Programmierung einer Anwendung (wie schnell arbeitet diese Routine ?)

Die Stoppuhr braucht am Programmende nicht wieder freigegeben werden !

Ergebnis: Long - 0

ReadWatch ()

Ermittelt die Anzahl der Millisekunden, die seit dem letzten Aufruf von StartWatch() vergangen sind.
Die Stoppuhr braucht am Programmende nicht wieder freigegeben werden !

Ergebnis: Long - Anzahl der Millisekunden, die seit dem letzten Aufruf von StartWatch() vergangen sind.

Beispiel:

```
StartWatch()
...
...                              ' hier steht Profancode
...
x&=Read Watch()

print Str$(x&)+" Millisekunden sind jetzt vergangen."
```

WaitWatch (M)

Wartet, bis mindestens M Millisekundenn seit dem letzten Aufruf von StartWatch() vergangen sind.
Diese Funktion ist sehr gut geeignet, um schnelle und langsame Computer zu synchronisieren.

```
M : Long - Anzahl Millisekunden (= tausenstel Sekunden)
```

Ergebnis: Long - 0

CreateImage (V,B,H)

Erzeugt eine leere Bitmap.

```
V : Long - Profan Variable %HDC
B : Long - Breite der neuen Bitmap
H : Long - Höhe der neuen Bitmap
```

Ergebnis: Long - Handle eines Memory Device Context (im Speicher gehaltene Grafikdaten-Struktur, HDC). Bei Fehler 0.

Beispiel:

```
ExternHDC&=CreateImage(%HDC,100,40)                'Erzeugt eine leere, 100 x 40
Pixel große Bitmap                                 'im Speicher
```

Die Bitmap wird so lange im Speicher gehalten, bis sie durch FreeImage() wieder entladen wird. Das sollte spätestens am Programmende erfolgen, um die Resourcen wieder frei zu geben.
Momentan können bis zu 32 externe Bitmaps gleichzeitig im Speicher gehalten werden, wem das noch zu wenig ist, der melde sich bitte bei mir.
Diese Funktion sollte erst dann aufgerufen werden, wenn Profan schon ein Fenster geöffnet hat.

LoadFileImage (S)

Lädt ein Bild in den Arbeitsspeicher zur späteren Benutzung. Es können die Grafikformate BMP, JPG, PNG, GIF und WMF geladen werden.

```
S : String mit Namen einer Grafikdatei (.bmp, .jpg, .gif, .png, .wmf)
```

Ergebnis: Long - Handle eines Memory Device Context (im Speicher gehaltene Grafikdaten-Struktur, HDC). Bei Fehler 0.

Beispiel:

```
text$="Picture.jpg"
ExternHDC&=LoadFileImage(Addr(text$))
```

Die geladene Grafik oder Bitmap wird so lange im Speicher gehalten, bis sie durch FreeImage() wieder entladen wird. Das sollte spätestens am Programmende erfolgen, um die Resourcen wieder frei zu geben.
Momentan können bis zu 32 externe Bitmaps gleichzeitig im Speicher gehalten werden. Siehe auch LoadMemoryImage() und LoadResourceImage().

LoadMemoryImage (B,A)

Holt ein Bild in den Arbeitsspeicher zur späteren Benutzung. Alle Grafiken der Formate BMP, JPG, GIF, PNG und WMF werden aus einem Speicherbereich (z.B. aus einem Bereich) heraus geladen.

```
B : Bereichsvariable
A : Long - Anzahl Bytes im Bereich
```

Ergebnis: Long - Handle eines Memory Device Context (im Speicher gehaltene Grafikdaten-Struktur, HDC). Bei Fehler 0.

Beispiel:

```
text$="Picture.jpg"
lenght&=@FileSize(text$)
If lenght&>0
  Dim memory#,lenght&
  ReadFileFast(addr(text$),memory#,0,lenght&)
  ExternHDC&=LoadMemoryImage(memory#,lenght&)
  Dispose memory#
EndIf
```

Die geladene/geholte Grafik oder Bitmap wird so lange im Speicher gehalten, bis sie durch FreeImage() wieder entladen wird. Das sollte spätestens am Programmende erfolgen, um die Resourcen wieder frei zu geben.
Momentan können bis zu 32 externe Bitmaps gleichzeitig im Speicher gehalten werden.

LoadResourceImage (H,S)

Lädt ein Bild aus einer Resource-Datei in den Arbeitsspeicher zur späteren Benutzung. Die Formate BMP, JPG, GIF, PNG und WMF werden als Grafikdateien unterstützt. Eine Resource-Datei kann eine Dll sein oder ein Exeprogramm (z.B. das eigene), das mit Daten zusammen gelinkt wird.

```
H : Long - HInstance einer Exedatei oder das Handle einer geladenen Dll.
S : Long - Nummer der Resource oder String mit Namen der Resource
```

Ergebnis: Long - Handle eines Memory Device Context (im Speicher gehaltene Grafikdaten-Struktur, HDC). Bei Fehler 0.

Beispiele:

```
1)
dll&=usedll("Resource.dll")
text$="Bild1"
ExternHDC&=LoadResourceImage(dll&,addr(text$))

2)
ExternHDC&=LoadResourceImage(%HInstance,100)
```

Für S kann eine Zahl (Long) oder ein String (Resourcename) angegeben werden. Es kann sogar eine Zahl im String angegeben werden, dann muß das erste Zeichen im String ein # sein und der Rest wird als Zahl interpretiert, z.B. text$="#100". Wird für S ein String angegeben, dann muß er mindestens 3 Zeichen enthalten, Namen mit weniger Zeichen sind nicht gestattet.
Für H kann eine Dll oder der HInstance des Exeprogramm angegeben werden, auch ein fremdes Exeprogramm, das Resourcen enthält, kann angegeben werden, sofern der HInstance dieses Programms ermittelt wurde.
Wichtig: Bmp-Dateien in Resourcen können als Resourcetyp "BITMAP" oder "IMAGE" gespeichert sein.
Alle anderen Grafik-Dateien müssen als Resourcetyp "IMAGE" gespeichert sein ! So wird z.B. ein Jpg-Bild nicht gefunden, das als Typ "BITMAP" gespeichert wurde...
Die geladene Grafik oder Bitmap wird so lange im Speicher gehalten, bis sie durch FreeImage() wieder entladen wird. Das sollte spätestens am Programmende erfolgen, um die Resourcen wieder frei zu geben.
Momentan können bis zu 32 externe Bitmaps gleichzeitig im Speicher gehalten werden.

PaintImage (Z,X1,Y1,B,H,Q,X2,X3,K)

Kopiert Teile eines Bildes oder das ganze Bild auf den Bildschirm oder in ein anderes Bild.

```
Z  : Long - Handle der Zielbitmap, z.B. das Profan Hauptfenster
X1 : Long - linke Koordinate in der Zielbitmap
Y1 : Long - obere Koordinate in der Zielbitmap
B  : Long - Breite des Rechtecks, das kopiert werden soll
H  : Long - Höhe des Rechtecks, das kopiert werden soll
Q  : Long - Handel der Quellbitmap
X2 : Long - linke Koordinate in der Quellbitmap
Y2 : Long - obere Koordinate in der Quellbitmap
K  : Long - Kopiermodus
```

Ergebnis: Long - TRUE, bei Fehler FALSE.

Der Bildschirmausschnitt der Quellbitmap beginnend bei X1,Y1 der eine Breite von B Pixel und eine Höhe von H Pixel hat, wird an die Position X2,Y2 in die Zielbitmap kopiert. Der letzte Parameter bestimmt den Kopiermodus.
Als Handel für die Quell- und Zielbitmap kann das Profanfenster (%HDC) angegeben werden oder ein von LoadFileImage() oder CreateImage() erhaltener Wert.
Es kann auch der HDC der Profan-internen Speicherbitmap angegeben werden, so wird er ermittelt:

```
Startpaint -1
SpeicherHDC&=%HDC
EndPaint
```

Werte für den Kopiermodus sind:

```
-1      = Transparent (transparente Farbe = Pixel oben rechts des
Quellrechtecks)
 0      = Normal kopieren
 1      = Quelle und Ziel mit AND verknüpfen
 2      = Quelle und Ziel mit OR verknüpfen
 3      = Quelle und Ziel mit XOR verknüpfen
 4      = Ziel invertieren (Quelle wird nicht berücksichtigt)
 5 - 12 = weitere Verknüpfungstechniken
 13     = Schwarz (default) löschen
 14     = Weiß (default) löschen
```

Wird als Kopiermodus die -1 gewählt, dann wird die Farbe des oberen rechten Pixels des zu kopierenden Rechtecks der Quellbitmap als transparente Farbe gewählt, im Gegensatz zum originalen Profan-Transparentmodus wo das transparente Pixel unten links gewählt wird.
Alle Pixel, die die transparente Farbe besitzen, werden nicht mit kopiert. So ist es möglich, auch Grafiken auf den Bildschirm zu bringen, die nicht rechteckig sind, sonders eine ganz individuelle Form besitzen.

Beispiel:

```
text$="Picture.bmp"
exhdc&=LoadFileImage(addr(text$))                        'Bild in externe Bitmap
laden

breite& = GetWidthImage(exhdc&)                           'Breite und Hoehe des
Bilds
hoehe& = GetHeightImage(exhdc&)                           'ermitteln

PaintImage(%hdc,200,100,breite&,hoehe&,exhdc&,0,0,0)   'Das externe Bild ins
Hauptfenster an Position
                           'X=200 / Y=100 kopieren.

'PaintImage(%hdc2,200,100,breite&,hoehe&,exhdc&,0,0,0) 'Ggf. das externe Bild
auch in die Profan
                       'Restaurierungsbitmap (%hdc2) kopieren, dann wird
                       'unsere Grafik auch bei Fensterüberlagerungen
                       'wieder automatisch restauriert !

FreeImage(exhdc&)                                       'Externes Bild freigeben
```

SizeImage (Z,X1,Y1,B1,H1,Q,X2,Y2,B2,H2,K)

Kopiert Teile eines Bildes oder das ganze Bild auf den Bildschirm oder in ein anderes Bild. Die Zielgröße ist hierbei, im Gegensatz zu PaintImage(), aber variabel. Dafür sind es auch 2 Parameter mehr.

```
Z  : Long - Handle der Zielbitmap, z.B. das Profan Hauptfenster
X1 : Long - linke Koordinate der Zielbitmap
Y1 : Long - obere Koordinate der Zielbitmap
B1 : Long - Neue Breite
H1 : Long - Neue Höhe
Q  : Long - Handel der Quellbitmap
X2 : Long - linke Koordinate der Quellbitmap
Y2 : Long - obere Koordinate der Quellbitmap
B2 : Long - Breite des Rechtecks, das kopiert werden soll
```

```
H2 : Long - Höhe des Rechtecks, das kopiert werden soll
K  : Long - Kopiermodus
```

Ergebnis: Long - TRUE, bei Fehler FALSE.

Diese Funktion ist fast gleich mit PaintImage(). Aber die Kopie der Bitmap kann jede beliebige Größe annehmen (B2 und H2).
Und als Kopiermodus ist -1 nicht zuläßig !!!

```
SizeImage(%hdc,0,0,Neuebreite&,Neuehoehe&,ExHDC&,0,0,Breite&,Hoehe&,0)
```

RotateImage (Z,X1,Y1,B,H,Q,X2,Y2,W,K)

Dreht Teile eines Bildes oder das ganze Bild auf den Bildschirm oder in ein anderes Bild. Die Drehung erfolgt in Grad (0-359).

```
Z  : Long - Handle der Zielbitmap, z.B. das Profan Hauptfenster
X1 : Long - linke Koordinate der Zielbitmap
Y1 : Long - obere Koordinate der Zielbitmap
B  : Long - Breite des Rechtecks, das kopiert werden soll
H  : Long - Höhe des Rechtecks, das kopiert werden soll
Q  : Long - Handel der Quellbitmap
X2 : Long - linke Koordinate der Quellbitmap
Y2 : Long - obere Koordinate der Quellbitmap
W  : Long - Winkel (0-359 Grad)
K  : Long - Kopiermodus
```

Ergebnis: Long - 0

Der Bildschirmausschnitt der Quellbitmap beginnend bei X1,Y1 der eine Breite von B Pixel und eine Höhe von H Pixel hat, wird an die Position X2,Y2 in die Zielbitmap im Winkel W kopiert. Der letzte Parameter bestimmt den Kopiermodus.
Die gedrehte Grafik wird dabei automatisch zentriert, was sehr praktisch ist.
Für den Kopiermodus sind nur die Werte 0 und -1 zulässig. Wird als Kopiermodus die -1 gewählt, dann wird die Farbe des oberen rechten Pixels des zu kopierenden Rechtecks der Quellbitmap als transparente Farbe gewählt, im Gegensatz zum originalen Profan-Transparentmodus wo das transparente Pixel unten links gewählt wird.
RotateImage() arbeitet sehr langsam, weil es auf der Get/SetPixel()-Technik basiert. Hierbei ist GetPixel() die große Bremse, die Sinus- und Cosinus Passagen und die Umrechnung von Bogenmaß nach Winkelmaß in Assembler wurden durch Tabellen realisiert, was eine atemberaubende Geschwindigkeit verleiht, wegen GetPixel() aber nicht viel nützt.
Zum Vorberechnen von kleinen drehenden Sprite-Animationen im Programm z.B. ist RotateImage() aber sehr gut geeignet. Um große Bilder zu drehen sollte lieber das wesentlich schnellere Rotate() verwendet werden.

Beispiel:

```
text$="Picture.png"
exhdc&=LoadFileImage(addr(text$))                    'Bild laden

breite& = GetWidthImage(exhdc&)                       'Breite und Hoehe des
Bilds
hoehe& = GetHeightImage(exhdc&)                       'ermitteln

WhileLoop  0,359,1                                          'Bild  in  360
Schritten
  RotateImage(%HDC,200,100,breite&,hoehe&,exhdc&,0,0,&loop,0) 'einmal komplett
```

```
Wend
                                                                    'herumdre
hen
```

FreeImage (H)

Entlädt ein Bild aus den Arbeitsspeicher und gibt die Resourcen wieder frei.

`H : Long - Rückgabewert einer LoadFileImage() oder CreateImage() Grafikdaten-Struktur (HDC)`

Ergebnis: Long - Eins oder grösser, bei Fehler 0.

Beispiel:

```
FreeImage(exhdc&)
```

Spätestens am Programmende sollte diese Funktion für jede externe Grafikdaten-Struktur erfolgen, um die Resourcen wieder frei zu geben.

FreeAllImages ()

Entlädt alle Bilder aus den Arbeitsspeicher, die mittels LoadFileImage() oder CreateImage() geladen/erstellt wurden und gibt die Resourcen wieder frei.

Ergebnis: Long - Anzahl Bilder, die freigegeben wurden.

Spätestens am Programmende sollten HDC's / Bilder wieder freigegeben werden. Das passiert normalerweise mittels FreeImage(), aber zur Sicherheit und für die Bequemlichkeit wurde noch FreeAllImages() dazu genommen, was auf einen Rutsch alle Bilder wieder freigibt.

Curve (H,SX,SY,CX,CY,EX,EY)

Zeichnet eine Kurve in die Bitmap eines beliebigen HDC's.

```
H  : Long - HDC
SX : Long - Startpunkt, X-Achse
SY : Long - Startpunkt, Y-Achse
CX : Long - Controlpunkt (Mittelpunkt), X-Achse
CY : Long - Controlpunkt (Mittelpunkt), Y-Achse
EX : Long - EndPunkt, X-Achse
EY : Long - EndPunkt, Y-Achse
```

Ergebniss: 0

Eine Funktion, um eine beliebige Kurve zu zeichnen. Die Farbe der Kurve wird mit den üblichen Grafikbefehlen eingestellt. Vom Startpunkt aus verläuft die Kurve zum Controlpunkt, wird bis zu diesem Punkt abgerundet und geht dann weiter zum Endpunkt. H kann jeder beliebige HDC sein, auch %HDC.

```
Curve(%hdc,50,50,400,300,100,200)
```

GetWidthImage (H)

Ermittelt die Breite einer Bitmap, die mit LoadFileImage() geladen oder mit CreateExtImage() erzeugt wurde.

```
H : Long - Handle der Bitmap
```

Ergebnis: Long - Breite der Bitmap (des Bildes) in Pixeln, bei Fehler 0.

Beispiel:

```
text$="Picture.gif"
exhdc&=LoadFileImage(addr(text$))                    'Bild in externe Bitmap
laden

x&=GetWidthImage(exhdc&)                              'Breite ermitteln
Print "Die Breite von Bild Picture.bmp ist "+Str$(x&) 'und mit Print ausgeben
```

GetHeightImage (H)

Ermittelt die Höhe einer Bitmap, die mit LoadFileImage() geladen oder mit CreateExtImage() erzeugt wurde.

```
H : Long - Handle der Bitmap
```

Ergebnis: Long - Höhe der Bitmap (des Bildes) in Pixeln, bei Fehler 0.

Beispiel:

```
text$="Picture.gif"
exhdc&=LoadFileImage(addr(text$))                   'Bild  in  externe  Bitmap
laden

x&=GetHeightImage(exhdc&)                             'Höhe ermitteln
Print "Die Höhe von Bild Picture.bmp ist "+Str$(x&)  'und mit Print ausgeben
```

VarToLong4 (V,A,W1,W2,W3,W4)

Schreibt vier Long-Werte hintereinander in einen Bereich.

```
V  : Bereichsvariable
A  : Long - Offsetadresse
W1 : Long - Zahl oder Variable
W2 : Long - Zahl oder Variable
W3 : Long - Zahl oder Variable
W4 : Long - Zahl oder Variable
```

Ergebnis: Long - 0

Beispiel:

```
x&=111
y&=222
VarToLong4(bereich#,0,1465,x&,x&+y&,271678)
VarToLong4(bereich#,16,...)
```

```
VarToLong4(bereich#,32,...)
```

Ergebniss in Bereich#:

```
Adresse: 0      4      8      12      ...
Wert:    1465   111    333    271678  ...
```

VarToLong10 (V,A,W1,W2,W3,W4,W5,W6,W7,W8,W9,W10)

Schreibt zehn Long-Werte hintereinander in einen Bereich.

```
V  : Bereichsvariable
A  : Long - Offsetadresse
W1 : Long - Zahl oder Variable
W2 : Long - Zahl oder Variable
W3 : Long - Zahl oder Variable
W4 : Long - Zahl oder Variable
W5 : Long - Zahl oder Variable
W6 : Long - Zahl oder Variable
W7 : Long - Zahl oder Variable
W8 : Long - Zahl oder Variable
W9 : Long - Zahl oder Variable
W10: Long - Zahl oder Variable
```

Ergebnis: Long - 0

Beispiel:

```
x&=111
y&=222
VarToLong10(bereich#,0,1465,x&,x&+y&,27167,5,6,x&*7,y&,9,1*10)
VarToLong10(bereich#,40,...)
VarToLong10(bereich#,80,...)
```

Ergebniss in Bereich#:

```
Adresse: 0      4      8      12     16     20     24     28     32     36
    ...
Wert:    1465   111    333    27167  5      6      777    222    9      10
    ...
```

LongToAddrVar4 (V,A,W1,W2,W3,W4)

Schreibt vier Longs aus einem Bereich direkt in vier Variablen (Long). Funktioniert erst ab Profan 7.

```
V  : Bereichsvariable
A  : Long - Offsetadresse
W1 : Long - Adresse von Variable 1
W2 : Long - Adresse von Variable 2
W3 : Long - Adresse von Variable 3
W4 : Long - Adresse von Variable 4
```

Ergebnis: Long - 0

Beispiel mit diesen Long-Werten in Bereich#:

```
Adresse: 0      4      8      12      ...
Wert:    1465   111    333    271678  ...
```

```
LongToAddrVar4(bereich#,0,addr(x&),addr(y&),addr(z&),addr(a&))
```

Nach dieser Funktion ist x&=1465, y&=111, z&=333 und a&=271678

InitSpriteBackground (M,B,H,XO,YO,XR,YR)

Erzeugt einen scrollenden Hintergrund, wenn mit Sprites gearbeitet wird. Nur mit dieser Funktion ist es möglich Scrolling und Sprites-Funktionen gleichzeitig zu realisieren, PaintImage() würde die Sprites flackern lassen.

```
M  : Long - Scrollmodus: 0 = Nicht scrollen / 1 = Scrollen
B  : Long - Breite des Profanfensters
H  : Long - Höhe des Profanfensters
XO : Long - X-Offset zum Beginn der Hintergrundsgrafik
YO : Long - Y-Offset zum Beginn der Hintergrundsgrafik
XR : Long - Scrollen in X-Richtung um XR Pixel (negative Werte scrollen nach
rechts)
YR : Long - Scrollen in Y-Richtung um YR Pixel (negative Werte scrollen nach
oben)
```

Ergebniss: Long - 0

Diese Funktion für sich allein ist nutzlos und nur in Verbindung mit ProSpeed.dll Sprites zu verwenden.
Das Scrollen des Hintergrunds startet erst, wenn mindestens ein Sprite erzeugt wurde. InitSpriteBackground() sollte aber gestartet worden sein, bevor Sprites erzeugt wurden. Auch wenn schon Sprites existieren kann durch wiederholtes Aufrufen von InitSpriteBackground() die Art des Scrolling noch verändert werden.
Besonderheit: Alle Werte dieser Funktion können auch durch den Dummywert 888888 ersetzt werden. 888888 bedeutet, das an dieser Stelle kein neuer Wert eingesetzt wird und der Wert, der durch ein vorheriges InitSpriteBackground() gesetzt wurde, erhalten bleibt.
InitSpriteBackground() macht die Ausführung von Sprites langsamer.

Beispiele:

```
InitSpriteBackground(1,800,600,0,0,1,0)                          'Scrollt
innerhalb eines 800 x 600 grossen
                                'Fenster nach links
InitSpriteBackground(888888,888888,888888,80,20,888888,888888)   'Setzt nur XO
und YO auf neue Werte
InitSpriteBackground(888888,888888,888888,888888,888888,0,0)     'Scrolling
stoppen
InitSpriteBackground(0,0,0,0,0,0,0)                              'Scrolling
ausschalten
```

InitSprite (S,F,K1,K2,X,Y,B,H,AF,SA,GA,XP,YP,SX,SY)

Erstellt ein (transparentes) Sprite und bringt es animiert auf den Bildschirm, ohne das sich das Profan-Hauptprogramm noch um seine Darstellung kümmern muß. Das Sprite bleibt am Bildschirm aktiv, bis es durch eine der Delete-Funktionen wieder vom Bildschirm gelöscht wird. Diese Funktion hat sehr viele Parameter, dafür ist dann aber der weitere Umgang mit dem Sprite sehr einfach.

```
S  : Long - Handle einer Bitmap die mit LoadFileImage() erzeugt oder geladen
wurde. In ihr müssen sich die Sprite Grafiken befinden
F  : Long - Handle des Profanfensters (fast immer %HDC), welches die
Hintergrunds-Grafik beinhaltet
K1 : Long - Handle einer Bitmap die mit LoadFileImage() erzeugt oder geladen
wurde und die Hintergrunds-Grafik beinhaltet (Kopie 1)
K2 : Long - Handle einer Bitmap die mit LoadFileImage() erzeugt oder geladen
wurde und die Hintergrunds-Grafik beinhaltet (Kopie 2)
X  : Long - X Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
Y  : Long - Y Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
B  : Long - Breite des Sprites
H  : Long - Höhe des Sprites
AF : Long - Anzahl Sprites in der Animation. Die verschiedenen Spritedaten einer
Animation müßen nebeneinander liegen und alle die gleiche Breite und Höhe haben
SA : Long - Mit diesem Bild startet die Animation (beginnt ab 1 und nicht ab
Null).
GA : Long - Geschwindigkeit, mit der die Animation abläuft, je niedriger, desto
schneller (1=sehr schnell)
XP : Long - X Position, an der das Sprite in der Hintergrundsgrafik dargestellt
wird
YP : Long - Y Position, an der das Sprite in der Hintergrundsgrafik dargestellt
wird
SX : Long - Geschwindigkeit, mit der sich das Sprite auf der X-Achse bewegt.
Eine Bewegung wird aber erst durch MoveSprite() gestartet.
SY : Long - Geschwindigkeit, mit der sich das Sprite auf der Y-Achse bewegt.
Eine Bewegung wird aber erst durch MoveSprite() gestartet.
```

Ergebniss: Long - Handle dieses Sprites, bei Fehler 0

Diese Funktion ist die mächtigste der ProSpeed.dll. Mit nur einer Funktion wird ein Sprite erzeugt und bleibt bis zu seiner Eliminierung auf dem Bildschirm, animiert und transparent zum Hintergrund.
Die transparente Farbe jedes einzelnen Sprites wird durch das rechte obere Pixel in der einzelnen Spritegrafik bestimmt.
Die verschiedenen Animations-Grafiken müssen nebeneinander in der Spritebitmap liegen und die gleiche Breite und Höhe haben.
Es können beliebig viele Sprites erzeugt werden, allerdings wird die Prozessor-Belastung mit jedem Sprite grösser. Man muß austesten, wieviele Sprites mit welchem Prozessor ruckelfrei bewegt werden können.
Kopie1 und 2 des Hintergrunds (K1/K2) werden für das Doubble-Buffering benötigt, unser Sprite soll ja nicht flackern.
Wenn durch die Funktion InitSpriteBackground() ein Hintergrund-Scrolling gestartet wurde, benötigt InitSprite() wesentlich mehr Prozessorzeit.
Die Werte XP und YP beschreiben die Position des Sprites innerhalb der Hintergrundgrafik, nicht innerhalb des Profan-Fensters. So kann es durchaus vorkommen, das ein Sprite momentan gar nicht zu sehen ist, weil der Bildschirm gerade nicht das entsprechenden Teil der Hintergrundsgrafik anzeigt, in dem sich das Sprite befindet. Ab Version 1.6 gibt es die Funktion SpriteScrollMode(), die das Bewegungsverhalten des Sprites abhängig von der Fensterposition machen kann.
SX und SY müßen immer positiv sein. Negative Werte sind nicht erlaubt. Eine Änderung der Bewegungsrichtung für das Sprite erreichen sie nicht durch andere (oder negative) Geschwindigkeitswerte, sondern nur durch andere Ziel-Positionswerte (z.b. mit MoveSprite()).
Vor dem Ende ihres Programmes müßen alle Sprites wieder gelöscht werden z.B. durch DeleteSprite(), zur Sicherheit sollten sie auch noch DeleteAllSprites() verwenden.

Beispiel:

```
Sprite1&=InitSprite(SpriteHdc&,
%HDC,KopieHdc1&,KopieHdc2&,0,0,24,18,4,1,8,100,100,1,1)
```

Erzeugt ein 24 x 18 Pixel großes Sprite an Postition 100/100. Animiert wird es gemächlich in vier Schritten.

Ausführlicheres Beispiel:

```
Declare
neu&,text$,background&,fensterkopie1&,fensterkopie2&,sprite1&,spriteplane&

$I Prospeed_Funktionen.inc  'Alle ProSpeed.dll Funktionen einbinden

' *** ProSpeed.dll öffnen
'
neu&=usedll("ProSpeed.dll")

text$="Hintergrund.jpg"
background&=LoadFileImage(addr(text$))  'Hintergrundbild laden (hier 800 x 600
Pixel groß)

WindowStyle 26
Windowtitle "Sprites Demo mit ProSpeed.dll."
Window 0,0-800,600

' *** Die Hintergrundsgrafik auf den Bildschirm kopieren
'
PaintImage(%HDC,0,0,800,600,background&,0,0,0)

' *** Die Hintergrundsgrafik in die profaninterne Kopie des Bildschirms kopieren
' *** So wird das Fenster wieder erneuert, wenn zeitweise Teile durch andere
Fenster verdeckt wurden.
'
PaintImage(%HDC2,0,0,800,600,background&,0,0,0)

fensterkopie1&=CreateImage(%HDC,800,600)  'Zwei Hdc's für Doublebuffering
fensterkopie2&=CreateImage(%HDC,800,600)  'anlegen

' *** Zwei Kopien des Bildschirms für das Sprite-Doublebuffering machen
'
PaintImage(fensterkopie1&,0,0,800,600,%HDC,0,0,0)
PaintImage(fensterkopie2&,0,0,800,600,%HDC,0,0,0)

text$="Sprites.gif"
spriteplane&=LoadFileImage(addr(text$))  'Spritegrafiken laden (nie Jpg-Grafiken
hierzu verwenden !!!)

sprite1&=InitSprite(spriteplane&,
%HDC,fensterkopie1&,fensterkopie2&,0,0,24,18,4,1,8,100,100,1,1)

WhileNot ende%
 WaitInput
 Case %key=2:BREAK
Wend

DeleteAllSprites()
```

```
FreeImage(background&)
FreeImage(spriteplane&)
FreeImage(fensterkopie1&)
FreeImage(fensterkopie2&)

freedll neu&
End
```

InitSprite5 (B)

Eine Profan 5 kompatible Lösung, um Sprites zu erstellen.

```
B : Long - Bereich, in dem die 15 Sprite-Parameter hintereinander (als Long-Int)
abgelegt sind
```

Ergebniss: Long - Handle dieses Sprites, bei Fehler 0

Profan 5 kann nur bis zu 12 Parameter an eine Dll übergeben, InitSprite() besitzt aber 15 Parameter. Darum ist InitSprite5() die einzige Möglichkeit, Sprites in "kleinen" Profanversionen zu benutzen.
Welche Parameter benötigt werden, ist bei InitSprite() nachzulesen.
Der Speicherbereich B muß mit mindestens 64 Bytes dimensioniert sein.

Beispiel mit InitSprite():

```
sprite&=InitSprite(spriteplane&,
%HDC,fensterkopie1&,fensterkopie2&,0,0,24,18,4,1,8,100,100,1,1)
```

Beispiel mit InitSprite5():

```
Long bereich#,00=spriteplane&
Long bereich#,04=%HDC
Long bereich#,08=fensterkopie1&
Long bereich#,12=fensterkopie2&
Long bereich#,16=0
Long bereich#,20=0
Long bereich#,24=24
Long bereich#,28=18
Long bereich#,32=4
Long bereich#,36=1
Long bereich#,40=8
Long bereich#,44=100
Long bereich#,48=100
Long bereich#,52=1
Long bereich#,56=1
sprite&=InitSprite5(bereich#)
```

AttachSprite (H,ZH,XO,YO)

Sprite H versucht automatisch an die Position zu gelangen, an der der sich Sprite ZH momentan befindet, mit Abstand XO, YO.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
ZH : Long - Ziel-Handle eines mit InitSprite() erzeugten Sprites
XO : Long - X Offset, X-Abstand zum Zielsprite
```

```
YO : Long - Y Offset, Y-Abstand zum Zielsprite
```

Ergebniss: Long - 0

Mit dieser Funktion lassen sich tolle Spriteeffekte verwirklichen, und das bei geringem User-Aufwand.
Um z.B. einen Wurm darzustellen, können mehrere Sprite (vielleicht Bälle) mit AttachSprite() verkettet werden. Hierbei jagt Sprite 1 dem Sprite 2 nach. Sprite 3 hinter Sprite 2 her, 4 hinter 3, usw, alle mit geringem Abstand. Jetzt braucht nur noch Sprite 1 bewegt werden, und die anderen Sprites bewegen sich automatisch wurmartig hinterher. Es sollten aber dann alle Sprites die gleiche Geschwindigkeit haben.
Um das Attach für Sprite H wieder aufzuheben, muß für ZH eine Null angegeben werden:

```
AttachSprite(attachsprite&,sprite&,10,10)
...
AttachSprite(attachsprite&,0,0,0) 'Attachmodus wieder beenden
```

ChangeSpritePara (H,P,V)

Verändert die verschiedenen Einstellungen eines Sprites manuell (nur für Profis).

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
P : Long - Offset des Parameters
V : Long - Neue Variable für Parameter P
```

Ergebniss: Long - 0

Für V werden LongInteger erwartet. Hier eine Liste einiger Parameter, bei denen neue Variablen sinnvoll wären:

```
Offset   Bedeutung
0        HDC der Spritebitmap, dort wo die Spritedaten "lagern"
4        X Offset in der Sprite HDC
8        Y Offset in der Sprite HDC
56       HDC des Profan-Fensters
68       HDC von Double-Buffer Kopie 1
84       HDC von Double-Buffer Kopie 2
92       Modus Hintergrund, 0=nicht scrollen / 1=scrollen
96       Hintergrund um X Pixel scrollen
100      Hintergrund um Y Pixel scrollen
156      Sprite Markierungs-Nummer
312      Startpunkt für MoveSpriteWithTable()
```

Beispiel, um einem Sprite neue Bitmap-Daten aus einer anderen HDC zuzuweisen inclusive neuer Offsets:

```
ChangeSpritePara(handle&,0,newhdc&)
ChangeSpritePara(handle&,4,newx&)
ChangeSpritePara(handle&,8,newy&)
```

CopySprite (H)

Kopiert das Sprite mit dem Handle H mit identischen Werten.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - bei Fehler 0, ansonsten Handle des neuen Sprites

Diese Funktion macht aus einem Sprite zwei.

Beispiel:

```
new&=CopySprite(old&)
```

MoveSprite (H,X,Y)

Bewegt das Sprite mit dem Handle H zu seinen neuen Koordinaten.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
X : Long - Neue X Ziel-Position des Sprites
Y : Long - Neue Y Ziel-Position des Sprites
```

Ergebniss: Long - 0

Das Sprite mit dem Handle H wird sich jetzt selbstständig zu den neuen Koordinaten begeben, völlig unabhängig vom Profan Programm.
Die Geschwindigkeit des Sprites für seine Fortbewegung wurde ja bereits in InitSprite() eingestellt, kann aber durch SetSpriteSpeed() verändert werden..
Die Animation des Sprites läuft weiterhin.
Es gibt zwei Bewegungs-Modi, siehe SetSpriteMovingMode().

```
MoveSprite(sprite&,400,300)
```

SetSpriteSpeed (H,X,Y)

Setzt für das Sprite mit dem Handle H eine neue Bewegungs-Geschwindigkeit.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
X : Long - Neue Geschwindigkeit, mit der sich das Sprite auf der X-Achse bewegt.
Y : Long - Neue Geschwindigkeit, mit der sich das Sprite auf der Y-Achse bewegt.
```

Ergebniss: Long - 0

X und Y müßen immer positiv sein. Negative Werte sind nicht erlaubt. Eine Änderung der Bewegungsrichtung für das Sprite erreichen sie nicht durch andere (oder negative) Geschwindigkeitswerte, sondern nur durch andere Ziel-Positionswerte (z.b. mit MoveSprite()).

```
SetSpriteSpeed(sprite&,4,4)
```

SetSpriteAnim (H,X,Y,B,H,AF,SA,GA)

Setzt für das Sprite mit dem Handle H eine neue Animation.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
X  : Long - X Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
Y  : Long - Y Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
```

```
B  : Long - Breite des Sprites
H  : Long - Höhe des Sprites
AF : Long - Anzahl Sprites in der Animation. Die verschiedenen Spritedaten einer
Animation müßen            nebeneinander liegen und alle die gleiche Breite und
Höhe haben
SA : Long - Mit diesem Bild startet die Animation (beginnt ab 1 und nicht ab
Null).
GA : Long - Geschwindigkeit, mit der die Animation abläuft, je niedriger, desto
schneller (1=sehr schnell)
```

Ergebniss: Long - 0

Weitere Infos über die Animation der Sprites unter InitSprite().

```
SetSpriteAnim(sprite&,0,0,24,18,4,1,8)
```

SetSpriteMovingMode (H,M)

Setzt für das Sprite mit dem Handle H einen anderen Bewegungs-Modus (normal oder Linienmodus).

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
M : Long - Modus, 0=normal / 1=direkte Linie
```

Ergebniss: Long - 0

Bisher bewegten sich Sprites zu ihrem neuen Zielpunkt, indem sie so schnell wie möglich die X- und Y-Achse des neuen Punktes erreichten. Meistens entstand hieraus eine Winkel-Flugbahn. Von den Koordinaten 0,0 nach 100,50 z.B. (bei gleicher Geschwindigkeit beider Achsen), wird das Sprite zuerst nach unten rechts fliegen. Als erstes erreicht es die Y-Bildschirm.Position. Von nun an fliegt es den Rest der Strecke nur noch nach rechts.
Ab ProSpeed 2.1 haben sie die Möglichkeit, Sprites so zum Zielpunkt gelangen zu lassen, das nur eine Linie entstehen würde, also direkter Flugweg.
Im Modus 1 wird als Geschwindigkeit für X und Y nur der Wert für X benötigt, also SetSpriteSpeed(handle&,4,1) würde für ein Sprite die Geschwindigkeit 4 einstellen.

Einige Einschränkungen hat der Linienmodus allerdings:
1) Es sollte kein direktes Setzen auf eine andere Position erfolgen (mit SetSpritePos).
2) Die Geschwindigkeit sollte nicht über 16 liegen

Der Linien-Modus basiert auf dem Bresenham Linien-Algorhythmus, eine sehr schnelle Methode um gerade Linien zu berechnen. Trotzdem verbraucht so eine Berechung natürlich etwas mehr Systemzeit als der normale Bewegungs-Modus. Die Unterschiede sind aber wirklich sehr gering.

```
SetSpriteMovingMode(sprite&,1)
```

SetSpriteAnimMove (H,R,X,Y)

Legt eine Animation für eine bestimmte Bewegungs-Richtung für das Sprite mit dem Handle H fest.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
R : Long - Richtung, in die sich das Sprite bewegt.
X : Long - X Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
Y : Long - Y Position (Offset) in der Spritebitmap, an der die gewünschten
Sprite Grafiken zu finden sind
```

Ergebniss: Long - 0

Eine neue mächtige Funktion.
Für jede Richtung, in die sich ein Sprite bewegt, kann jetzt eine eigene Animations-Folge vorgegeben werden. So können Sprites erzeugt werden, die sich unabhängig vom Hauptprogramm in jede Richtung bewegen und immer unterschiedlich animiert sind, z.B. eine Person, die am Bildschirm herum läuft.
Für die unterschiedlichen Richtungen sind folgende Zahlen vorgesehen.

```
0 = Stillstand
1 = rechts oben
2 = rechts
3 = rechts unten
4 = unten
5 = links unten
6 = links
7 = links oben
8 = oben
```

Die Anzahl und der Ablauf der verschiedenen Animationschritte werden nicht verändert, nur X und Y Offset in der Spritebitmap werden neu gesetzt. Durch Aufruf von SetSpriteAnim() werden alle Bewegungsphasen wieder gelöscht.

Beispiel:

```
kraehe&=InitSprite(Sprites&,
%HDC,Fensterkopie1&,Fensterkopie2&,0,192,64,64,2,1,16,600,0,2,1)
SetSpriteAnimMove(kraehe&,0,0,0)
SetSpriteAnimMove(kraehe&,1,192,64)
SetSpriteAnimMove(kraehe&,2,0,64)
SetSpriteAnimMove(kraehe&,3,192,192)
SetSpriteAnimMove(kraehe&,4,0,128)
SetSpriteAnimMove(kraehe&,5,192,128)
SetSpriteAnimMove(kraehe&,6,0,192)
SetSpriteAnimMove(kraehe&,7,192,0)
SetSpriteAnimMove(kraehe&,8,0,0)
```

SetSpriteAnimMode (H,M)

Legt fest, ob eine Animation immer wieder von vorne gespielt wird (loop) oder mit der letzten Animations-Phase stoppt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
M : Long - Modus der Animation. 0 = von vorne / 1 stoppen
```

Ergebniss: Long - 0

Wenn sie als Modus 0 angeben, fängt die Animation des Sprites immer wieder von vorne an. Geben sie jedoch 1 an, dann stoppt die Animation mit dem letzten Bild.
Ob die Animation schon gestoppt hat, können sie durch GetSpriteAnimModeStatus() abfragen.
Diese Funktion eignet sich hervoragend z.B. für explodierende Raumschiffe. Die Explosions-Animation lassen sie im Modus 1 ablaufen und das letzte Bild ist dann ein leeres Sprite. Sobald GetSpriteAnimModeStatus() erkannt hat, das die Explosion zuende ist, löschen sie einfach das Sprite mit DeleteSprite().
Der voreingestellte Modus für Animationen ist immer 0 (loop).

```
SetSpriteAnimMode(sprite&,0)
```

SetSpritePos (H,X,Y,XN,YN)

Setzt für das Sprite mit dem Handle H eine neue Position am Bildschirm.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
X  : Long - Neue X-Position, an die das Sprite gesetzt (nicht bewegt) wird
Y  : Long - Neue Y-Position, an die das Sprite gesetzt (nicht bewegt) wird
XN : Long - Neue X-Position, zu der das Sprite als nächstes bewegen wird
YN : Long - Neue Y-Position, zu der das Sprite als nächstes bewegen wird
```

Ergebniss: Long - 0

Diese Funktion setzt das Sprite an neue Koordinaten und bewirkt gleichzeitig ein MoveSprite(). Wenn sie nur neue Koordinaten wünschen, dann setzten sie X und XA und ebenfalls Y und YA auf identische Werte.

```
SetSpritePos(sprite&,30,200,30,200)
```

SetSpriteCage (H,X1,Y1,X2,Y2,XN,YN)

Prüft automatisch, ob sich ein Sprite aus einem rechteckigen Bereich heraus bewegt hat, wenn ja, wird das Sprite an eine neue Position gesetzt und seine Bewegung gestoppt.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
X1 : Long - Linke Position des rechteckigen Bereichs
Y1 : Long - Obere Position des rechteckigen Bereichs
X2 : Long - Rechte Position des rechteckigen Bereichs
Y2 : Long - Untere Position des rechteckigen Bereichs
XN : Long - Neue X-Position, an die das Sprite gesetzt wird, wenn das Rechteck
verlassen wurde
YN : Long - Neue Y-Position, an die das Sprite gesetzt wird, wenn das Rechteck
verlassen wurde
```

Ergebniss: Long - 0

Diese Funktion wird innerhalb des Sprite-Threads (also automatisch) ausgeführt. Jeden Durchlauf wird dann getestet, ob das Sprite mit dem Handle H sich aus dem unter X1, Y1, X2, Y2 angegebenen Rechteck heraus bewegt hat. Wenn ja wird es an seine neue Position XN, YN gesetzt und seine Bewegung gestoppt. Um diese Funktion wieder ungültig zu machen, müßen X1, Y1, X2, Y2 mit Null angegeben werden, z.B. mit SetSpriteCage(handle&,0,0,0,0,0,0).
Ein sinnvoller Einsatz wäre es, wenn Raumschiffe Schüsse abfeuern. SetSpriteCage() begrenzt dann die Schüsse auf den sichtbaren Teil des Bildschirms. Geraten die Schüsse aus dem Bildschirm, wird dann ihre Position automatisch auf z.B. -40,-40 gesetzt. Jetzt könnte ein DeleteAllXSprites(-40) oder ein DeleteAllYSprites(-40) im Profanprogramm automatisch alle Schüsse löschen, die nicht mehr sichtbar sind.

```
SetSpriteCage(sprite&,0,0,1024,768,-40,-40)
```

SetSpriteFixMode (H,M)

Setzt den Zeichenmodus für das Sprite mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
M : Long - Zeichenmodus, 0 = normal / 1 = Unbewegtes, nicht animiertes Sprite
```

Ergebniss: Long - 0

Im Normalfall ist dieser Modus 0. Sie können ihn aber mit dieser Funktion auf 1 setzten. Dadurch wird das Sprite nicht mehr neu gezeichnet, macht also quasi eine Pause und verbraucht keine Systemzeit mehr. SetSpriteFixMode() ist nur für Sprites interessant, die sich nicht bewegen, nicht animiert sind und nicht von anderen Sprites überlagert werden. Oder für Sprites, die gerade nicht sichtbar sind !
Alle anderen Funktionen können aber immer noch für dieses Sprite angewandt werden, z.B. die Kollisionsfunktionen. Sobald ein anderes Sprite unser "fixiertes" Sprite berührt, können sie es mit `SetSpriteFixMode(sprite&,0)` wieder "zum Leben erwecken" oder mit DeleteSprite() löschen.
Ein Anwendung wäre z.B. ein Breakout Spiel, wo viele Spielsteine unbewegt darauf warten, bis ein Ball sie abschießt.
Siehe auch GetSpriteFixMode() und VisualTableFix().

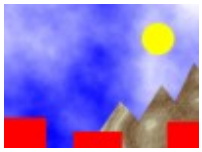
SetBackAutoCollision (H,M,F,P1,P2,P3,P4,P5,P6,P7,P8)

Automatische Sprite-Hintergrunds Kollisions-Abfrage mit vielen Einstellungsmöglichkeiten.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
M  : Long - Handle einer mit LoadFileImage() geladenen speziellen Masken-Bitmap
F  : Long - Flag
P1 : Long - 1. Parameter (genutzt von einigen Flags, ansonsten 0)
P2 : Long - 2. Parameter (genutzt von einigen Flags, ansonsten 0)
P3 : Long - 3. Parameter (genutzt von einigen Flags, ansonsten 0)
P4 : Long - 4. Parameter (genutzt von einigen Flags, ansonsten 0)
P5 : Long - 5. Parameter (genutzt von einigen Flags, ansonsten 0)
P6 : Long - 6. Parameter (genutzt von einigen Flags, ansonsten 0)
P7 : Long - 7. Parameter (genutzt von einigen Flags, ansonsten 0)
P8 : Long - 8. Zeiger auf einen Sound, der mit DSoundLoad() geladen wurde,
ansonsten 0)
```

Ergebniss: Long - 0

Eine neue und mächtige Funktion. Endlich ist eine richtige Kollision eines Sprites mit dem Hintergrund machbar, und die reagiert auch noch automatisch, ohne das das Hauptprogramm belastet wird...
In Parameter 8 kann (unabhängig vom Flag) ein Zeiger auf eine Buffersammlung stehen, in die mit DSoundLoad() eine Wave-Datei (.WAV) geladen wurde. Bei einer Kollision wird dann ein Soundeffekt gespielt.
In jedem Fall benötigen wir für M eine spezielle Maskenbitmap, am besten eine Masken-Kopie des Hintergrunds. Diese sollte zweifarbig sein (1 Bit schwarz-weiß Bild).
Alle Stellen im Hintergrund, die von dem Sprite begehbar sein sollen, müssen in der Farbe weiss gezeichnet sein.
Alle anderen Stellen im Hintergrund, die das Sprite nicht betreten darf, müssen schwarz gekennzeichnet sein. In diesem Fall findet eine Hintergrundskollision statt und das Sprite kann nicht dort hingelangen (ab ProSpeed 2.6 wurden die beiden Farben getauscht, um kompatibel zu sein mit Maskenbitmaps für andere ProSpeed-Funktionen, z.B. FindPath()).




Hintergrundsbitmap (links): Die drei Mauern dürfen nicht vom Sprite durchquert werden, alles andere wohl.
Maskenbitmap (rechts): Alles was weiss ist kann durchquert werden, die schwarzen Flächen (stehen für die Mauern) nicht.

Für F stehen mehrere Flags zur Auswahl, die nicht kombinierbar sind.

```
0 = Keine Auto-Kollision mit diesem Sprite überprüfen.
1 = Sprite bleibt stehen, sobald es eine nicht passierbare Fläche berührt.
2 = Sprite wird wie ein Gummiball in eine andere logische Richtung gelenkt.
3 = Wie 2, aber Sprite hat einen ständigen Drang, nach oben zu gelangen
(Hoppelball nach oben)
4 = Wie 2, aber Sprite hat einen ständigen Drang, nach rechts zu gelangen
(Hoppelball nach rechts)
5 = Wie 2, aber Sprite hat einen ständigen Drang, nach unten zu gelangen
(Hoppelball nach unten)
6 = Wie 2, aber Sprite hat einen ständigen Drang, nach links zu gelangen
(Hoppelball nach links)
7 = Sprite durchläuft eine neue Animation und wird danach gelöscht
```

Für die folgenden Flags müssen sie gültige Parameter (P1-P4) angeben:

Flags 2 bis 6:

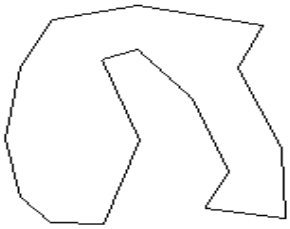
```
P1 = linke Ecke eines Movepunkts zu dem das Sprite bei einer Umkehr gelangen
will (am besten 0).
P2 = obere Ecke eines Movepunkts zu dem das Sprite bei einer Umkehr gelangen
will (am besten 0).
P3 = rechte Ecke eines Movepunkts zu dem das Sprite bei einer Umkehr gelangen
will.
P4 = untere Ecke eines Movepunkts zu dem das Sprite bei einer Umkehr gelangen
will.
```

Flag 7 (siehe auch SetSpriteAnim):

```
P1 = X Position (Offset) in der Spritebitmap, an der die gewünschten Sprite
Grafiken zu finden sind
P2 = Y Position (Offset) in der Spritebitmap, an der die gewünschten Sprite
Grafiken zu finden sind
P3 = Breite des Sprites
P4 = Höhe des Sprites
P5 = Anzahl Sprites in der Animation. Die verschiedenen Spritedaten einer
Animation müßen      nebeneinander liegen und alle die gleiche Breite und Höhe
haben
P6 = Mit diesem Bild startet die Animation (beginnt ab 1 und nicht ab Null).
P7 = Geschwindigkeit, mit der die Animation abläuft, je niedriger, desto
schneller (1=sehr schnell)
```

Beispiel für eine Gummiball-Kollisions-Abfrage (Flag 2) mit Sound:

```
SetBackAutoCollision(sprite&,maske&,2,0,0,GetWidthImage(maske&),GetHeightImage(m
aske&),0,0,0,buffer1&)
```



*Bei den ProSpeed Demo's gibt es den Sourcecode **FA_SetBackAutoCollision.prf**, der diese Funktion sehr gut demonstriert. Vier Sprites (Bälle) sausen durch einen labyrinth-artigen Hintergrund.*
Die Bälle benutzen für F den Flag 2. Auf eine zusätzliche Sprite-Sprite-Kollision wurde aus Übersichtsgründen verzichtet.
Mit dem Sourcecode kann sehr leicht experimentiert werden.

CountSprites ()

Zählt, wieviele Sprites momentan neugezeichnet werden / aktiv sind.

Ergebniss: Long - Anzahl Sprites

CountAllSprites ()

Zählt, wieviele Spritetabellen existieren. Mitgezählt werden auch alle bis dato gelöschten Sprites !

Ergebniss: Long - Anzahl Spritetabellen

ExistSprite (H)

Testet, ob das Sprite mit dem Handle H überhaupt noch existiert.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0 = ja, existiert noch / -1 = nein, das Sprite wurde gelöscht (Delete)

Weil ja einige Funktionen automatisiert sind, kann es sein, das sie mal wissen wollen, ob ein Sprite überhaupt noch existiert. Das können sie hiermit erfahren.
Aber selbst wenn sie aus Versehen mit einem Spritehandle arbeiten, das es gar nicht mehr gibt (z.B. beim Einsatz von CollideMore), ist das egal, es hat dann nur keine Auswirkungen. Die interne Spritestruktur bleibt noch solange bestehen, bis das letzte Sprite gelöscht wurde oder CleanUp() benutzt wurde, erst dann werden alle Strukturen aus dem Speicher entfernt.

```
ExistSprite(sprite&)
```

HasSpriteArrived (H)

Testet, ob das Sprite mit dem Handle H schon an seinen Ziel-Koordinaten angekommen ist.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0 = nicht angekommen / 1 = am Zielpunkt angekommen

Wenn sie ein Sprite mit MoveSprite() oder SetSpritePos() zu neuen Koordinaten geschickt haben, dann testet diese Funktion, ob es schon angekommen ist.

```
HasSpriteArrived(sprite&)
```

SpriteMovingRect (H,X1,Y1,X2,Y2)

Errichtet eine rechteckige Begrenzung für das Sprite mit dem Handle H, innerhalb der sich das Sprite automatisch bewegt. Stößt es an eine Grenze, dann ändert es selbstständig seine Bewegungs-Richtung.

```
H  : Long - Handle eines mit InitSprite() erzeugten Sprites
X1 : Long - Linke Begrenzung des Sprites
Y1 : Long - Obere Begrenzung des Sprites
X2 : Long - Rechte Begrenzung des Sprites
Y2 : Long - Untere Begrenzung des Sprites
```

Ergebniss: Long - 0

Eine mächtige neue Funktion, zusammen angewendet mit SetSpriteAnimMove() hervoragend geeignet fürJump&Run-Spiele.
Wird für X1, Y1, X2 oder Y2 eine -1 angegeben, dann wird die Begrenzung für diese Richtung wieder aufgehoben. Das muß dann aber immer paarweise geschehen, also immer gleichzeitig für X1 und X2 oder gleichzeitig für Y1 und Y2. Nur einen der vier Werte mit -1 zu belegen ist nicht zuläßig und führt zu Fehlern!

```
SpriteMovingRect(sprite&,0,0,1024,768) 'setzt die Begrenzung auf 0/0/1024/768
...
SpriteMovingRect(sprite&-1,0,-1,768)   'hebt die horizontale Begrenzung wieder
auf
SpriteMovingRect(sprite&,-1,-1,-1,-1)  'hebt die horizontale und vertikale
Begrenzung wieder auf
```

SpriteScrollMode (H,M)

Ändert das Bewegungverhalten eines Sprites bei aktivem Bildschirmscrolling.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
M : Long - Modus, 0 = normal, die Spritekoordinaten sind relativ zur Nullachse
der Hintergrundsgrafik.
        1 = neu, die Spritekoordinaten sind relativ zur Nullachse des
Programmfensters.
```

Ergebniss: Long - 0

Im Modus 1 lassen sich jetzt auch Sprites realisieren, die absolut unbeweglich am Bildschirm stehen bleiben können, egal, wie sehr der Hintergrund auch gescrollt wird.

```
SpriteScrollMode(sprite&,1)
```

SpriteToHDC (H)

Kopiert die aktuelle Grafik des Sprites mit dem Handle H auf den Bildschirm (und in die beiden Bildschirm-Kopien).

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0

Eine Kopie des Sprite wird bei den aktuellen Koordinaten fest auf dem Bildschirm verankert und ist fortan Teil des Hintergrunds. Das Sprite an sich reagiert aber weiter wie gewohnt.

```
SpriteToHDC(sprite&)
```

SpriteThreadBrake (M)

Wartet nach jedem Sprite, das auf den Bildschirm kopiert wurde, M Millisekunden. Hiermit läuft das Profan-Hauptprogramm wesentlich schneller, der Sprite-Thread wird natürlich langsamer. Für bewegte Sprites nicht unbedingt ratsam, um Sprites als Icons zu benutzen (auf Buttons, Toolleiste usw.) aber sehr sinnvoll.

```
M : Long - Anzahl Millisekunden (empfohlen: 1-10)
```

Ergebniss: Long - 0

Wird für M Null verwendet, dann wird die Bremswirkung wieder aufgehoben.

```
SpriteThreadBrake(4)
...
SpriteThreadBrake(0)
```

MouseOverSprite (B,A,X,Y)

Prüft, ob sich der Mauszeiger über einem Sprite befindet.

```
B : Long - Bereichsvariable, in der eine Liste von Spritehandles (Longs) steht.
A : Long - Anzahl von Spritehandles, die in Bereich B stehen.
X : Long - Profanvariable %mousex
Y : Long - Profanvariable %mousey
```

Ergebniss: Long - 0, bei Fehler oder keiner Berührung, ansonsten Handle des ersten berührenden Sprites aus dem Bereich B, das den Mauszeiger momentan berührt.

X und Y bezeichnen üblicherweise die linke obere Ecke des Mauszeigers.
Diese Funktion eignet sich ganz hervoragend für Spiele a la Moorhuhn.

Beispiel:

```
VarToLong4(bereich#, 0,sprite1&,sprite2&,sprite3&,sprite4&)
VarToLong4(bereich#,16,sprite5&,sprite6&,sprite7&,sprite8&)
VarToLong4(bereich#,32,sprite9&,sprite10&,0,0)               '10 Spritehandle in
Bereich schreiben

x&=MouseOverSprite(bereich#,10,%mousex,%mousey)               'testen, ob eines
der                    Sprites                      gerade                    den
                                                            'Mauszeiger
berührt.

CaseNot   @Equ(x&,0):DeleteSprite(x&)                          'Moorhuhn   würde
jetzt das Sprite löschen
```

MoveSpriteWithTable (H,B,A,X,Y,S)

Setzt für ein Sprite einen festgelegten Bewegungs-Pfad. Die X- und Y-Werte hierfür werden aus einer vorher generierten Tabelle ausgelesen.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
B : Long - Bereichsvariable, in der in Tabellenform eine Liste von Bildschirm-
Koordinaten steht (oder 0).
A : Long - Anzahl von Koordinatenpaaren in der Tabelle B
X : Long - X Offset, der zu jeder X-Koordinate hinzugefügt wird (darf auch
negativ sein)
Y : Long - Y Offset, der zu jeder Y-Koordinate hinzugefügt wird (darf auch
negativ sein)
```

```
S : Long - Startkoordinatenpaar, 1 bis ..., ab wo das Auslesen aus der Tabelle
beginnt
```

Ergebniss: Long - 0

Bisher konnte einem Sprite nur ein Punkt auf dem Bildschirm angegeben werden, zu dem es sich bewegen soll.
Ab ProSpeed 2.1 kann ein Sprite jede beliebige Bewegung am Bildschirm vollziehen. Hierzu wird aus der Tabelle B ein Koordinatenpaar ausgelesen, zu dem sich das Sprite als nächstes bewegt. Ist es dort angekommen, wird das nächste Koordinatenpaar ausgelesen usw.
Ist das Ende der Tabelle erreicht, dann wird wieder vom Startpunkt der Tabelle weiter ausgelesen. Soll die Tabelle nur einmal abgearbeitet werden, dann können sie diesen Einmal-Modus mit SpriteTableMode() einstellen oder auch wieder zurückstellen.
Die Werte in der Tabelle müßen als Integer (Word) gespeichert sein, immer ein Word für X und dann eines für Y, dann X, Y, X, Y, ...
Um so eine Tabelle zu generieren ist der *Pathfinder* von Moritz Kersten sehr geeignet, der dem ProSpeed-Paket als Freeware beiliegt:
Aber auch mit der Funktion PixelToMemory() kann aus einer Grafik eine Koordinatentabelle generiert werden.
Ein Beispiel für den Einsatz solcher Sprite-Bewegungen wäre z.B. ein Jump-and-Run Spiel oder ein Ballerspiel a la *RType*.

Ist B = 0, dann verhält sich das Sprite wieder normal, es fliegt noch bis zu den zuletzt angegebenen Koordinaten bleibt dann stehen.
Die Tabelle in B wird übrigens von ProSpeed kopiert und braucht nicht in der Bereichsvariablen gespeichert bleiben.

Beispiel, um ein Sprite immer um den Bildschirm fliegen zu lassen:

```
Text$="0,0,1024,0,1024,768,0,768"                  'Koordinatenwerte
SetWords(Bereich#,0,Addr(Text$))                   'als Word-Integer in Bereich#
eintragen

sprite&=InitSprite(...)
MoveSpriteWithTable(sprite&,bereich#,4,0,0,1)      'Und ab geht's
```

Möchte man die Flugbahn des Sprites umkehren (es soll also die Tabellenwerte in umgekehrter Reihenfolge abfliegen), dann bietet sich die Funktion ChangeLongSequence() an, um alle Koordinatenwerte umzukehren. Das obere Beispiel müßte dann so lauten:

```
Text$="0,0,1024,0,1024,768,0,768"                  'Koordinatenwerte
SetWords(Bereich#,0,Addr(Text$))                   'als Word-Integer in Bereich#
eintragen
ChangeLongSequence(bereich#,16)                    'Tabellenwerte umkehren
(Sprite fliegt anders herum)

sprite&=InitSprite(...)
MoveSpriteWithTable(sprite&,bereich#,4,0,0,1)      'Und ab geht's
```

GetSpritesFps ()

Ermittelt die Geschwindigkeit des Sprite Threads in Frames per second (Fps).

Ergebniss: Long - Fps des Sprite Threads

Der zurückgegebene Wert ermittelt, wie oft pro Sekunde die Sprites neu gezeichnet werden.

GetSpriteX (H)

Ermittelt die X-Position des Sprite mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - X-Position innerhalb der Hintergrundsgrafik.

```
GetSpriteX(sprite&)
```

GetSpriteY (H)

Ermittelt die Y-Position des Sprite mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Y-Position innerhalb der Hintergrundsgrafik.

```
GetSpriteY(sprite&)
```

GetSpriteMoveX (H)

Testet, ob sich das Sprite mit dem Handle H momentan nach links oder rechts bewegt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Bewegung des Sprites auf der X-Achse

```
-1 = Sprite bewegt sich nach links
 0 = Sprite bewegt sich nicht nach links und nicht nach rechts
 1 = Sprite bewegt sich nach rechts
```

```
GetSpriteMoveX(sprite&)
```

GetSpriteMoveY (H)

Testet, ob sich das Sprite mit dem Handle H momentan nach oben oder unten bewegt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Bewegung des Sprites auf der Y-Achse

```
-1 = Sprite bewegt sich nach oben
 0 = Sprite bewegt sich nicht nach oben und nicht nach unten
 1 = Sprite bewegt sich nach unten
```

```
GetSpriteMoveY(sprite&)
```

GetSpriteMoveXY (H)

Testet, in welche Richtung sich das Sprite mit dem Handle H momentan bewegt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Bewegungsrichtung des Sprites.

```
0 = Sprite steht still
1 = Bewegung nach rechts oben
2 = Bewegung nach rechts
3 = Bewegung nach rechts unten
4 = Bewegung nach unten
5 = Bewegung nach links unten
6 = Bewegung nach links
7 = Bewegung nach links oben
8 = Bewegung nach oben
```

```
GetSpriteMoveXY(sprite&)
```

GetSpriteInfos (H,B)

Ermittelt einige wichtige Daten für das Sprite mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
B : Bereichsvariable
```

Ergebniss: Long - 0 = Fehler / ansonsten B.

Die ermittelten Daten werden als Longs in den bereitgestellten Bereich geschrieben. Dieser Bereich muß mindestens 32 Bytes groß sein.
Folgende Daten werden in den Bereich geschrieben, hier dargestellt als Offset-Tabelle.

```
 0 = aktuelle X-Position
 4 = aktuelle Y-Position
 8 = Zielposition X (eingestellt durch MoveSprite(), SetSpritePos())
12 = Zielposition Y (eingestellt durch MoveSprite(), SetSpritePos())
16 = Ist Sprite schon am Zielpunkt angekommen (0=nein / 1=ja) ?
20 = eingestellte Geschwindigkeit, mit der sich das Sprite auf der X-Achse
bewegt.
24 = eingestellte Geschwindigkeit, mit der sich das Sprite auf der Y-Achse
bewegt.
28 = aktuelle Animationsphase
```

Beispiel:

```
Dim bereich#,32
GetSpriteInfos(sprite&,bereich#)

x&=Long(bereich#,16)  'Beispielabfrage: Ist Sprite schon am Zielpunkt angekommen
?
If x&
   ...
EndIf
```

```
Dispose bereich#
```

GetSpriteAnimModeStatus (H)

Prüft, ob eine Sprite-Animation im Modus 1 schon am Ende angekommen ist.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Status

```
0 = Animation läuft noch
1 = Animation hat schon gestoppt
```

Diese Funktion ist nur im Zusammenhang mit SetSpriteAnimMode() zu gebrauchen. Nähere Infos entnehmen sie bitte dort.

```
x&=GetSpriteAnimModeStatus(sprite&)
If x&=1
   ...
Else  'wenn x&=0
   ...
EndIf
```

GetSpriteFixMode (H)

Ermittelt den Zeichenmodus für das Sprite mit dem Handle H, der mit SetSpriteFixMode() gesetzt wurde. Siehe auch VisualTableFix().

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Zeichenmodus: 0 = normal / 1 = als unbewegtes, nicht animiertes Sprite gekennzeichnet

```
x&=GetSpriteFixMode(sprite&)
```

GetSpriteMark (H)

Ermittelt die Markierung des Sprites mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Zahl, mit der das Sprite markiert wurde.

Ermittelt die Markierung eines Sprite, die mit MarkSprite() vergeben wurde. So können Sprites eines bestimmten Zahlen-Typs erkannt werden, sehr sinnvoll nach einer Kollision.
Hierdurch kann bestimmt werden, das nur bestimmte Sprites bei einer Kollision mit z.B. CollideAll() reagieren sollen.

```
x&=GetSpriteMark(sprite&)
```

GetBackgroundInfos (B)

Ermittelt einige wichtige Daten über das Hintergrunds-Scrolling, das zusammen mit Sprites gestartet werden kann (InitSpriteBackground()).

```
B : Bereichsvariable
```

Ergebniss: Long - B.

Die ermittelten Daten werden als Longs in den bereitgestellten Bereich geschrieben. Dieser Bereich muß mindestens 28 Bytes groß sein.
Folgende Daten werden in den Bereich geschrieben, hier dargestellt als Offset-Tabelle.

```
 0 = Scrollmodus, 0=scrollt nicht / 1=scrollt
 4 = eingestellte Fensterbreite
 8 = eingestellte Fensterhöhe
12 = X-Offset zum Beginn der Hintergrundsgrafik
16 = Y-Offset zum Beginn der Hintergrundsgrafik
20 = X Richtung, scrollen um X Pixel (kann auch negativ sein)
24 = Y Richtung, scrollen um Y Pixel (kann auch negativ sein)
```

Beispiel:

```
Dim bereich#,28
GetBackgroundInfos(bereich#)

x&=Long(bereich#,8)  'Beispielabfrage: Eingestellte Fensterhöhe
If x&
   ...
EndIf

Dispose bereich#
```

GetSpriteLayer (H)

Ermittelt den Layer (Darstellungs-Ebene) eines Sprite.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Nummer der Ebene, bei Fehler -1.

Jedes Sprite hat eine andere Darstellungsebene (Layer). Je niedriger diese ist, desto früher wird das Sprite auf den Bildschirm gebracht, sodaß es von den darauffolgenden Sprites verdeckt werden kann.
Die Layer reichen von 1 bis Anzahl Sprites.

```
x&=GetSpriteLayer(sprite&)
```

SwapSpriteLayers (H1,H2)

Vertauscht die Darstellungs-Ebenen (Layer) von zwei Sprites.

```
H1 : Long - Sprite 1, erster Handle eines mit InitSprite() erzeugten Sprites
H2 : Long - Sprite 2, zweiter Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0, wenn ein Fehler stattfand -1.

Die Layer der beiden Sprites H1 und H2 werden vertauscht. Ein sinnvoller Einsatz wäre z.B.:
Eine Figur läuft hinter einem Baum vorbei, dann geht sie zurück und läuft diesmal vor dem Baum her. So kann ein räumliches Spielfeld geschaffen werden.
Wichtig! Das Sprite, welches als erstes generiert worden ist, darf nicht geswappt werden. Mit anderen Worten, das Sprite mit dem Wert GetSpriteLayer(...)=1 darf nicht vertauscht werden. Es besitzt einen Sonderstatus.

```
SwapSpriteLayers(sprite1&,sprite2&)
```

MarkSprite (H,Z)

Markiert das Sprite mit dem Handle H mit der Zahl Z.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
Z : Long - Zahl
```

Ergebniss: Long - 0

Markiert ein Sprite. So können nachher Sprites eines bestimmten Zahlen-Typs erkannt werden, z.B. nach einer Kollision.
Siehe auch GetSpriteMark(). Sehr sinnvolle Funktion...

```
MarkSprite(sprite&,111)
```

Collision (H1,H2,X,Y)

Testet, ob gerade zwei Sprites zusammenstossen.

```
H1 : Long - Sprite 1, erster Handle eines mit InitSprite() erzeugten Sprites
H2 : Long - Sprite 2, zweiter Handle eines mit InitSprite() erzeugten Sprites
X  : Long - X Ausdehnung der Pixelabtastung
Y  : Long - Y Ausdehnung der Pixelabtastung
```

Ergebniss: Long - Status der Kollision oder bei Fehler 0

Für Ergebniss gibt es diese beiden Möglichkeiten:

```
0 = momentan findet keine Kollision dieser beiden Sprites statt oder ein es lag
ein Fehler vor
1 = die beiden Sprites berühren sich gerade
```

Diese Funktion wurde für ProSpeed V2.1 komplett neu erstellt und arbeitet jetzt mit anderer Technik. Zwei Parameter sind hinzugekommen, um die Dauer dieser Funktion herabzusetzten. Bislang war Collision() sehr langsam, das lag an der pixelgenauen Abtastung und mit 0/0 in den Parameter X und Y ist die Funktion immer noch genauso langsam.
X und Y geben an, wieviele Pixel bei der Collisions-Abtastung immer übersprungen werden, somit müßen weit weniger Pixel überprüft werden. Ein Aufruf von Collision(sprite1&,sprite2&,4,4) z.B. geht sehr schnell und ist dabei noch sehr sicher in der Kollisionsabfrage. Um die optimalen Werte für ihre beiden Sprites heraus zu finden, sollten sie einfach ein bischen herumexperimentieren.
Eine Beschränkung auf bestimmte Grafikmodi gibt es jetzt für Collision nicht mehr.

```
x&=Collision(sprite1&,sprite2&,4,4)
```

Collide (H1,H2)

Testet, ob gerade zwei Sprites zusammenstossen.

```
H1 : Long - Sprite 1, erster Handle eines mit InitSprite() erzeugten Sprites
H2 : Long - Sprite 2, zweiter Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - Status der Kollision oder bei Fehler 0

Für Ergebniss gibt es diese beiden Möglichkeiten:

```
0 = momentan findet keine Kollision dieser beiden Sprites statt oder ein es lag
ein Fehler vor
1 = die beiden Sprites berühren sich gerade
```

Diese Funktion arbeitet nicht pixelgenau, sondern überprüft, ob die Bereiche (Breite x Höhe) der Sprites sich berühren. Dafür arbeitet sie blitzschnell. Wenn sie aber eine pixelgenaue Kollisionsroutine benötigen, dann nehmen sie besser Collision().

```
x&=Collide(sprite1&,sprite2&)
```

CollideMore (H,B,A)

Testet, ob das Sprite mit dem Handle H mit anderen Sprites gerade zusammenstößt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites.
B : Long - Bereichsvariable, in der eine Liste von Spritehandles (Longs) steht.
A : Long - Anzahl von Spritehandles, die in Bereich B stehen.
```

Ergebniss: Long - 0, bei Fehler oder keiner Kollision, ansonsten erster gefundener Spritehandle aus dem Bereich B, der mit H momentan kollidiert

Diese Funktion arbeitet blitzschnell, aber nicht pixelgenau. Dafür können sofort viele Sprites auf eine Kollision getestet werden, ohne das eine langsame Profan-Schleife hierfür verwendet werden müßte.

Beispiel:

```
VarToLong4(bereich#, 0,sprite1&,sprite2&,sprite3&,sprite4&)
VarToLong4(bereich#,16,sprite5&,sprite6&,sprite7&,sprite8&)
VarToLong4(bereich#,32,sprite9&,sprite10&,0,0)              '10 Spritehandle in
Bereich schreiben

x&=CollideMore(sprite&,bereich#,10)                         'testen, ob
sprite0& mit einem dieser
                              'Sprites kollidiert.
```

CollideAll (H)

Testet, ob das Sprite mit dem Handle H mit irgendeinem anderen Sprite zusammenstößt.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0 oder Handle des ersten gefundenen Sprites, das gerade kollidiert.

Diese Funktion arbeitet nicht pixelgenau, dafür ist sie aber eine schnelle Möglichkeit einer fast kompletten Kollisionsprüfung. Noch ausführlicher (und genauso schnell) ist nur CollideUnknown().
Dauert nicht mal 1 Millisekunde für hunderte von Sprites...
Sehr sinnvoll mit der Funktionen MarkSprite() zu benutzen.

```
x&=CollideAll(sprite&)
```

SpriteTableMode (H,M)

Stellt den Modus ein, mit dem MoveSpriteWithTable() arbeiten soll, entweder Einmal-Modus (M=1) oder Loop-Modus (M=0).

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites.
M : Long - Modus, 0 = normal / 1 = Tabelle nur einmal abarbeiten
```

Ergebniss: Long - 0

Voreingestellt ist der Loop-Modus, die Tabelle wird immer wieder von vorne abgearbeitet. Wenn Tabellen kominiert mit Wegsuche benutzt werden, ist der Einmal-Modus aber sinnvoller.

```
SpriteTableMode(sprite&,1)
```

StopAllSprites (S)

Stoppt alle Sprites und den scrollenden Hintergrund. Eine Pausefunktion eben.

```
S : Long - Schalter-Parameter: 0 = normaler Modus / 1 = alles stoppt
```

Ergebniss: Long - 0

Der Hintergrund hört auf zu scrollen und alle Sprites halten an, wenn als Parameter 1 angegeben wird. Mit 0 als Parameter geht alles wieder seinen gewohnten Gang.
Passen sie aber auf, den gleichen Parameter nicht mehrmals hintereinander aufzurufen. Also immer schön AUS-AN-AUS-AN-AUS-.....

```
StopAllSprites(1)
...
StopAllSprites(0)
```

DeleteSprite (H)

Löscht das Sprite mit dem Handle H.

```
H : Long - Handle eines mit InitSprite() erzeugten Sprites
```

Ergebniss: Long - 0

Löscht das Sprite vom Bildschirm. H darf fortan nicht mehr benutzt werden.

```
DeleteSprite(sprite&)
```

DeleteAllSprites ()

Löscht alle Sprites vom Bildschirm.

Ergebniss: Long - 0

Sollte am Ende eines Programmes immer benutzt werden, um sicher zu gehen, das auch alle Sprites gelöscht wurden. Ansonsten kann es beim Beenden des Profan-Programmes zu einer Fehlermeldung kommen.

DeleteSpritesIfAnimReady ()

Löscht alle Sprites vom Bildschirm, deren Animation im Modus 1 beendet wurde.

Ergebniss: Long - Anzahl der gelöschten Sprites.

Sehr sinnvoll im Zusammenhang mit SetSpriteAnimMode() und GetSpriteAnimModeStatus().
Optimal für z.B. Explosionen einsetzbar, Explosions-Sprite wird gelöscht, wenn die Animation durchgelaufen ist.

DeleteAllXSprites (X)

Löscht alle Sprites vom Bildschirm, die sich momentan an Bildschirm-Position X befinden.

```
X : Long - Bildschirm-Position in der X-Achse
```

Ergebniss: Long - Anzahl der gelöschten Sprites.

Kann sehr sinnvoll sein. So können bestimmte Sprites gelöscht werden, ohne deren Spritehandle zu kennen. Sprites die aus dem Bildschirm laufen usw.

```
DeleteAllXSprites(-40)
```

DeleteAllYSprites (Y)

Löscht alle Sprites vom Bildschirm, die sich momentan an Bildschirm-Position Y befinden.

```
Y : Long - Bildschirm-Position in der Y-Achse
```

Ergebniss: Long - Anzahl der gelöschten Sprites.

Kann sehr sinnvoll sein. So können bestimmte Sprites gelöscht werden, ohne deren Spritehandle zu kennen. Sprites die aus dem Bildschirm laufen usw.

```
DeleteAllYSprites(-40)
```

DeleteSpritesMark (Z)

Löscht alle Sprites vom Bildschirm, die mit der Zahl Z markiert wurden.

```
Z : Long - Zahl, mit der das Sprite durch MarkSprite() markiert wurde.
```

Ergebniss: Long - Anzahl der gelöschte Sprites.

Sehr sinnvoll im Zusammenhang mit MarkSprite() und GetSpriteMark().

```
DeleteSpritesMark(111)
```

CleanUp (M)

Räumt in den internen Spritetabellen auf. Jede Spritetabelle, deren Sprite die Markierung M hatte und (scheinbar) gelöscht wurde, wird hiermit komplett freigegeben.

```
M : Long - Markierung (für bestimmte Sprites)
```

Ergebniss: Long - 0

Jedes Sprite der ProSpeed.dll speichert seine Daten in einer Tabelle. Alle Tabellen sind hierbei miteinander verkettet. Wenn jetzt ein Sprite gelöscht wird, wird seine Tabelle als gelöscht gekennzeichnet, existiert aber noch so lange weiter, bis das letzte Sprite gelöscht wurde. Das ist auch gut so, sollte also noch auf ein Sprite zugegriffen werden, obwohl es schon gelöscht wurde, passiert einfach nichts. Wäre der komplette Tabellenspeicher für das Sprite ganz gelöscht, dann würde es ansonsten einen lustigen Absturz geben...
CleanUp() räumt richtig auf und löscht alle Tabellen mit der Markierung M, die nicht mehr benötigt werden. CleanUp() sollte trotzdem nicht sorglos benutzt werden, weil möglicherweise Funktionen wie CollideMore() danach falsche Werte erhalten können, z.B. Sprites, deren Speicher gar nicht mehr existiert. Wenn aber sehr viele Sprites initiiert und wieder gelöscht wurden, ist CleanUp() hin und wieder sehr sinnvoll und beschleunigt den ganzen Sprite-Thread wieder.
Wenn sie bei ihren Sprites ohne Markierung arbeiten, können sie für M Null angeben.
Ach ja, mehrere CleanUp()-Funktionen sollten nicht hintereinander stehen, weil sonst nur der erste abgearbeitet wird. Die Funktion arbeitet nämlich nicht sofort (kehrt natürlich aber sofort zum Hauptprogramm zurück), sondern erst an passender Stelle im Sprite-Thread. Eine mögliche Lösung für mehrer CleanUp() Aufrufe innerhalb einer kontinuierlichen Schleife wäre:

```
x&=Rnd(3)
Case x&=0:CleanUp(20)
Case x&=1:CleanUp(114)
Case x&=2:CleanUp(34)
```

ReadFileFast (S,B,O,A)

Schnelles Lesen einer Datei in einem Bereich.

```
S : Zeiger eines 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.
```

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

Schneller und bequemer kann man eine Datei wohl nicht mehr laden. Kein Assign, Filemode, Open und Close mehr.
Aus der Datei S werden A Bytes in den Bereich B + O geladen. Es wird im Modus "nur lesen" geladen.
Beispiel:

Anstatt...

```
text$="Dateiname"
Assign #2,text$
Filemode 0
OpenRW #2
BlockRead(#2,bereich#,0,bytes&)
CloseRW #2
```

...schreiben sie jetzt nur noch:

```
text$="Dateiname"
ReadFileFast(addr(text$),bereich#,0,bytes&)
```

ReadFilePieceFast (S,B,O,A,P)

Schnelles Lesen eines Teiles einer Datei in einem Bereich.

```
S : Zeiger eines 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.
P : Long - Anzahl Bytes, die von Beginn der Datei S übersprungen werden.
```

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

Beschreibung siehe ReadFileFast(), nur mit dem Unterschied, das ReadFilePieceFast() einen Teil in der Mitte oder am Ende einer Datei lesen kann. Vorne stehende Informationen können übersprungen werden. Man stelle sich vor, man möchte nur einige Bytes einer großen Datei lesen, aber die stehen ganz hinten in der Datei (MP3-Dateien z.B.). Warum dann die ganze Datei laden?

```
text$="Dateiname"
ReadFilePieceFast(addr(text$),bereich#,0,8,1600000)
```

Das Beispiel lädt 8 Bytes einer großen Datei in einen Bereich, wobei die ersten 1.6 MB übersprungen werden.

WriteFileFast (S,B,O,A)

Schnelles Schreiben einer Datei aus einem Bereich.

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

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

Schneller und bequemer kann man eine Datei wohl nicht mehr speichern. Kein Assign, Filemode, Open und Close mehr.
In die Datei S werden A Bytes aus dem Bereich B + O geschrieben. Es wird im Modus "nur schreiben" gespeichert. Wenn schon eine Datei des gleichen Names existiert, wird sie überschrieben.
Beispiel:

Anstatt...

```
text$="Dateiname"
Assign #2,text$
Filemode 1
OpenRW #2
BlockWrite #2,bereich#,0,bytes&
CloseRW #2
```

...schreiben sie jetzt nur noch:

```
text$="Dateiname"
WriteFileFast(addr(text$),bereich#,0,bytes&)
```

CreateFX (B,H)

Erzeugt manuell eine Byte-Array Struktur.

```
B : Long - Breite des imaginären Bildes
H : Long - Höhe des imaginären Bildes
```

Ergebniss: Long - Handle einer Byte-Array Struktur, bei Fehler 0.

Eine Byte-Array Struktur wird angelegt. Die Grafikdaten (Pixeldaten), also das eigentliche Byte-Array, ist mit 0 Werten gefüllt (schwarzes Bild).
Nähere Erläuterungen unter InitFX(). Siehe auch PngToFX().

```
CreateFX(1024,768)
```

InitFX (H)

Erzeugung einer Bytes-Array Struktur aus einer externen Bitmap. Nötig für alle grafischen Effekte.

```
H : Long - HDC einer mit LoadFileImage(), LoadMemoryImage(), LoadResourceImage()
oder CreateImage() erzeugten Grafikdaten-Struktur
```

Ergebniss: Long - Handle einer Byte-Array Struktur, bei Fehler 0.

Neu ab Version 1.7 sind die grafischen Effekte. Bevor diese jedoch angewendet werden können, muß (!) eine spezielle Bitmap (als Byte-Array) hierfür erzeugt werden (bei einigen Effekten sogar mehrere). Hierzu muß in H ein HDC übergeben werden, und zwar eine mit ProSpeed erzeugte, keinesfalls darf %HDC übergeben werden.
Mit InitFX() wird eine spezielle Kopie von H erzeugt, die später direkt manipuliert werden kann. Der Inhalt von H wird durch die grafischen Effekten nicht verändert.
Ganz ganz wichtig: Die Breite der Bitmap H muß durch vier teilbar sein !!! Das liegt daran, wie Windows diese Bitmaps verwaltet. Wenn die Breite nicht durch vier teilbar wäre (Windows integriert ansonsten zusätzliche Füll-Bytes), dann würde der Verwaltungsaufwand sehr viel höher sein und die Geschwindigkeit der grafischen Effekte wäre wesentlich niedriger. Ich glaube, das ist die Sache wert.
Also, immer Breite durch vier teilbar, bei der Höhe spielt es keine Rolle. Bitte beachten sie das, ansonsten gibt es bei einigen FX-Funktionen einen lustigen Absturz...
Jedes Byte-Array muß spätestens zum Programmende mit FreeFX() wieder freigegeben werden, um den Speicher wieder frei zu machen Bei einer Bildgrösse von 1024x768 wären das immerhin 2.359.296 Bytes die sonst verloren gingen.

Siehe auch CreateFX() und PngToFX().

Beispiel:

```
exhdc&=CreateImage(breite&,hoehe&)              'Leere Bitmap in der Größe des
                    'Bildschirms erzeugen
PaintImage(exhdc&,0,0,breite&,hoehe&,%HDC,0,0,0) 'Inhalt des Profanfensters
hierhin kopieren
array&=InitFX(exhdc&)                          'Handle (Byte-Array) zu einer
Bitmap                                                  'erzeugen
...
FreeFX(array&)                                 'später wieder freigeben
```

FreeFX (B)

Freigabe von Speicher, der mit PngToFX(), InitFX() oder CreateFX() erzeugt wurde.

```
B : Long - Handle eines durch PngToFX(), InitFX() oder CreateFX() erzeugten Byte-
Arrays
```

Ergebniss: Long - 0.

Nähere Erläuterungen unter InitFX(), PngToFX() und CreateFX().

```
FreeFX(array&)
```

SaveFX (B,S,F)

Speichert ein Byte-Array als Bitmap-Grafikdatei ab, bisher sind Bilder aber nur als Bmp-Dateien speicherbar.

```
B : Long - Handle eines durch PngToFX(), InitFX() oder CreateFX() erzeugten Byte-
Arrays
S : Long - Zeiger auf einen String mit dem Dateinamen der zu speichernder Datei.
F : Long - Flags für Dateiformat
```

Ergebniss: Long - Anzahl Bytes, die in die Datei geschrieben wurde, bei Fehler 0.

Jetzt ist es auch mit Profan möglich Bilder abzuspeichern, die größer sind als maximal 2000 x 2000 Pixel.
Alle Bilder bis zu einer Größe von 32768 x 32768 sind speicherbar.
Als Flags gibt es bisher:

```
0 = Bmp-Datei
```

Beispiel:

```
text$="Bild.bmp"

array&=InitFX(exhdc&)                          'Handle   (Byte-Array)   zu   einer
Bitmap erzeugen
SaveFX(array&,Addr(text$),0)
FreeFX(array&)                                 'später wieder freigeben
```

SaveToMemoryFX (B,M,F)

Kopiert ein Byte-Array als Bitmap-Grafikdatei in einem Speicherbereich. Quasi die Umkehrfunktion zu LoadMemoryImage().

```
B : Long - Handle eines durch PngToFX(), InitFX() oder CreateFX() erzeugten Byte-
Arrays
M : Long - Zeiger auf einen Speicherbereich, in den die Daten kopiert werden
sollen.
F : Long - Flags für Dateiformat
```

Ergebniss: Long - Anzahl Bytes, die nach M kopiert wurden, bei Fehler 0.

Nähere Erläuterungen bitte unter SaveFX() nachlesen. Der Unterschied zwischen SaveFX() und SaveToMemoryFX() ist, das hier die Daten nicht in einer Datei gespeichert werden, sondern in einem Speicherbereich.

Beispiel:

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

x&=GetWidthImage(exhdc&)            'Breite und Hoehe des Hdc's exhdc&
y&=GetHeightImage(exhdc&)           'ermitteln und Speicher für SaveToMemoryFX()
Dim bereich#,(x&*3*y&)            'reservieren: Breite * 3 * Höhe

SaveToMemoryFX(array&,bereich#,0)
FreeFX(array&)                 'später wieder freigeben
```

Darken (F,X,Y,B,S)

Abdunklung des Bildschirms (oder Teilen davon) bis hin zu einer schwarzen Fläche.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
S : Long - Schrittweite der Abdunklung
```

Ergebniss: Long - 0.

Der Bildschirm wird um S Stufen verdunkelt. S darf hierbei von 1 bis 255 liegen. Der aktuelle Grafikmodus ist für diese Funktion egal. Intern wird mit 24 Bit gerechnet.
X und Y geben die linke obere Ecke an, ab der auf dem Bildschirm verdunkelt wird. Die Grösse der zu verdunkelnden Zone ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Darken() ist ziemlich schnell und arbeitet in Echtzeit. Grosse Bildschirme (z.B. 1024x768) sollten auf langsamen Rechnern aber nicht in 1-Stufen-Schritten abgedunkelt werden. Ab einem 600er Prozessor sollte die Geschwindigkeit hierfür aber reichen.
Generell gilt: Je kleiner die Fläche, desto schneller wird sie abgedunkelt.
Bevor Darken() angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

Beispiel:
Ein Bildschirm wird langsam verdunkelt:

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Darken(%HDC,0,0,array&,1)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

Lighten (F,X,Y,B,S)

Aufhellung des Bildschirms (oder Teilen davon) bis hin zu einer weißen Fläche.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
S : Long - Schrittweite der Aufhellung
```

Ergebniss: Long - 0.

Der Bildschirm wird um S Stufen erhellt. S darf hierbei von 1 bis 255 liegen. Der aktuelle Grafikmodus ist für diese Funktion egal. Intern wird mit 24 Bit gerechnet.
X und Y geben die linke obere Ecke an, ab wo der Bildschirm erhellt wird. Die Grösse der aufzuhellenden Zone ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Lighten() ist ziemlich schnell und arbeitet in Echtzeit. Grosse Bildschirme (z.B. 1024x768) sollten auf langsamen Rechnern aber nicht in 1-Stufen-Schritten erhellt werden. Ab einem 600er Prozessor sollte die Geschwindigkeit hierfür aber reichen.
Generell gilt: Je kleiner die Fläche, desto schneller wird sie erhellt.
Bevor Lighten() angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

Beispiel:
Ein Bildschirm wird langsam erhellt:

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Lighten(%HDC,0,0,array&,1)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

Blur (F,X,Y,B)

Verwischt die Grafik auf dem Bildschirm (oder Teile davon).

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
```

B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-Arrays (Quellbitmap)

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird verwischt. X und Y geben die linke obere Ecke an, ab wo der Bildschirm verwischt wird. Die Grösse des zu verwischenden Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Blur() ist ziemlich schnell und arbeitet in Echtzeit. Ab einem 600er Prozessor sollte auch ein großer Bildschirm flüssig verwischt werden.
Generell gilt: Je kleiner die Fläche, desto schneller arbeitet Blur().
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Blur(%HDC,0,0,array&)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

Rustle (F,X,Y,B)

Erzeugt ein Rauschen auf dem Bildschirm (oder auf Teilen davon) ähnlich dem eines Fernsehers ohne Sendeempfang.

F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-Arrays (Quellbitmap)

Ergebniss: Long - 0.

Die Grafik des Bildschirms rauscht. X und Y geben die linke obere Ecke an, ab wo der Bildschirm rauschen soll. Die Grösse des rauschenden Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Rustle() ist ziemlich schnell und arbeitet in Echtzeit. Generell gilt: Je kleiner die Fläche, desto schneller arbeitet Rustle().
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Rustle(%HDC,0,0,array&)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

Grey (F,X,Y,B)

Macht bunte Bildschirme (oder Teile davon) grau. Tauscht also alle Farben in Grautöne um.

```
F : Long - Handle der Zielbitmap (HDC) erzeugte) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird in Grautönen abgebildet. X und Y geben die linke obere Ecke an, ab wo der Bildschirm verändert wird. Die Grösse des graufarbigen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Grey() ist ziemlich schnell und arbeitet in Echtzeit. Generell gilt: Je kleiner die Fläche, desto schneller arbeitet diese Funktion.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert.
Noch ein Wort zu den Grautönen. Davon gibt es maximal 256 verschiedene *richtige* Grautöne, nämlich 0/0/0, 1/1/1, 2/2/2, ..., 254/254/254, 255/255/255. Das gilt es zu beachten.

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
Grey(%HDC,0,0,array&)
FreeFX(array&)                 'später wieder freigeben
```

SemiTrans (F,X,Y,B1,B2,P)

Mischt zwei verschiedene Bilder zu einem, hierbei wird durch eine Prozentangabe die Transparenz jedes Bildes berücksichtigt (semi-transparente Einblendung).

```
F  : Long - Handle der Zielbitmap (HDC) oder 0
X  : Long - X Offset zum Bildschirm
Y  : Long - Y Offset zum Bildschirm
B1 : Long - Erstes Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays (1. Quellbitmap)
B2 : Long - Zweites Handle eines mit PngToFX(), InitFX() oder CreateFX()
erzeugten Bytes-Arrays (2. Quellbitmap)
P  : Long - Prozentangabe, nur Werte von 0 bis 100 verwenden.
```

Ergebniss: Long - 0.

Die Grafiken der Bilder B1 und B2 werden unter Berücksichtigung der Prozentangabe P nach F kombiniert. Es sieht dann so aus, als wären beide Grafiken transparent. Der Hintergrund wird aber hierdurch gelöscht. Wenn sie den Hintergrund zusammen mit einem Bild kombinieren wollen, so müssen sie mit einer entsprechenden Kopie des Hintergrunds arbeiten, welche für B1 oder B2 eingesetzt wird.
X und Y geben die linke obere Ecke an, an die beide Bilder gesetzt werden. Die Grösse des einzublendenen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Die Grösse von B1 und B2 muß aber immer gleich groß sein !!!
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber

nicht auf den Bildschirm gebracht.
SemiTrans() ist nicht ganz so schnell wie die anderen Grafikfilter, arbeitet aber mit einem 800 MHz Computer auch bei großen Bildschirmen in Echtzeit. Generell gilt: Je kleiner die Fläche, desto schneller arbeitet diese Funktion.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten in den Byte-Arrays B1 und B2 werden durch Aufruf dieser Funktion nicht verändert. Vielmehr wird intern mit einem dritten Speicherbereich gearbeitet. SemiTrans() wird also in einer Schleife nur Wirkung zeigen, wenn sie die Prozentangabe ändern.
Noch zu beachten ist der eingestellte Windows-Grafikmodus. So hat nicht jede Steigerung um 1 Prozent Auswirkungen in jedem Modus.
Z.B. im 16 Bit Modus wird nur alle 6.25 Prozent eine Änderung der Farbwerte vorgenommen, weil nur 16 verschiedene Abstufungen für jeden Farbwert verwendet werden können, 16x16x16 für Rot-Grün-Blau. Nur der 32 Bit-Modus (und auf einigen alten Grafikkarten der 24 Bit-Modus) können in 1 Prozent Stufen überblendet werden. Das ist für ihre Programmierung aber egal, funktionieren wird es auf allen Grafik-Systemen, nur ist nicht jede Abstufung überall sichtbar.

Beispiel:
Ein Bild wird ganz soft in die vorhandene Hintergrundsgrafik eingeblendet.

```
fxhandle1&=CreateExtImage(%hdc,640,480)                          'Handle 1: leere
Bitmap erzeugen
name$="Bild.bmp"
fxhandle2&=LoadFileImage(addr(name$))                           'Handle 2: Bild
laden, z.B. 640x480 groß
PaintImage(fxhandle1&,0,0,640,480,%HDC,0,0,0)                    'Inhalt des
Profanfensters nach Handle 1
bytearray1&=InitFX(fxhandle1&)                                 'Byte-Array zu
Handle 1 erzeugen
bytearray2&=InitFX(fxhandle2&)                                 'Byte-Array zu
Handle 2 erzeugen

WhileLoop 100
 SemiTrans(%HDC,0,0,bytearray2&,bytearray1&,&loop)              'Das Bild wird in
100 Schritten in die
Wend
                                                             'Hintergrundgrafik
eingeblendet

FreeFX(bytearray1&)                                            'Byte-Array 1 wieder
freigeben
FreeFX(bytearray2&)                                            'Byte-Array 2 wieder
freigeben
FreeImage(fxhandle1&)                                           'Handle 1 wieder
freigeben
FreeImage(fxhandle2&)                                           'Handle 2 wieder
freigeben
```

ColorMove (F,X,Y,B1,B2,S)

Mischt zwei verschiedene Bilder zu einem, anders als bei SemiTrans() werden aber nicht zwei Bilder transparent vermischt, sondern es werden nur die Farben beider Bilder angenähert. Diese Funktion ist wesentlich schneller als SemiTrans() und es können auch mit einem Trick Bilder eingeblendet werden, die nicht nur rechteckig sind, Sprites z.B.

```
F  : Long - Handle der Zielbitmap (HDC) oder 0
X  : Long - X Offset zum Bildschirm
Y  : Long - Y Offset zum Bildschirm
```

```
B1 : Long - Erstes Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays (1. Quellbitmap)
B2 : Long - Zweites Handle eines mit PngToFX(), InitFX() oder CreateFX()
erzeugten Bytes-Arrays (2. Quellbitmap)
S  : Long - Schrittweite der Annäherung.
```

Ergebniss: Long - 0.

Die Farben der Grafik B2 werden mit der Schrittweite S immer mehr den Farben der Grafik B1 angenähert. Die Byte-Daten von B2 werden hierdurch verändert. Wenn sie den Hintergrund zusammen mit einem Bild kombinieren wollen, so müssen sie mit einer entsprechenden Kopie des Hintergrunds arbeiten, welche für B1 oder B2 eingesetzt wird.
X und Y geben die linke obere Ecke an, an die beide Bilder gesetzt werden. Die Grösse des einzublendenen Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Die Grösse von B1 und B2 muß aber immer gleich groß sein !!!
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
ColorMove() ist superschnell und arbeitet aber mit einem 400 MHz Computer auch bei großen Bildschirmen in Echtzeit. Generell gilt: Je kleiner die Fläche, desto schneller arbeitet diese Funktion.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B1 werden durch Aufruf dieser Funktion nicht verändert, die von B2 schon.
Je größer die Schrittweite S ist, desto schneller wird eingeblendet. Hier gelten Werte von 1-255.

Beispiel:
Ein Sprite wird ganz soft in die vorhandene Hintergrundsgrafik eingeblendet.

```
fxhandle1&=CreateExtImage(%hdc,80,80)                              'Handle 1: leere
Bitmap erzeugen
PaintImage(fxhandle1&,0,0,80,80,%HDC,0,0,0)                        'Inhalt des
Profanfensters nach Handle 1
name$="Bild.bmp"
fxhandle2&=LoadFileImage(addr(name$))                               'Handle 2:
Sprite laden, z.B. 80x80 groß
PaintImage(fxhandle1&,0,0,80,80,fxhandle2&,0,0,-1)                 'Sprite im
transparenten Modus nach Handle 1
                                'kopieren
PaintImage(fxhandle2&,0,0,80,80,%HDC,0,0,0)                        'Inhalt des
Profanfensters nach Handle 2
bytearray1&=InitFX(fxhandle1&)                                   'Byte-Array zu
Handle 1 erzeugen
bytearray2&=InitFX(fxhandle2&)                                   'Byte-Array zu
Handle 2 erzeugen

WhileLoop 256
 ColorMove(%HDC,0,0,bytearray1&,bytearray2&,1)                     'Das Sprite wird
in 256 Schritten in die
Wend
                                                               'Hintergrundgrafik
eingeblendet

FreeFX(bytearray1&)                                              'Byte-Array 1 wieder
freigeben
FreeFX(bytearray2&)                                              'Byte-Array 2 wieder
freigeben
FreeImage(fxhandle1&)                                              'Handle 1 wieder
freigeben
FreeImage(fxhandle2&)                                              'Handle 2 wieder
freigeben
```

Sharpen (F,X,Y,B)

Verschärft die Farben der Grafik auf dem Bildschirm (oder Teile davon). Wie auch immer, jedenfalls der Umkehr-Effekt von Blur().

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms scheint schärfer zu werden. X und Y geben die linke obere Ecke an, ab wo der Bildschirm schärfer wird. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Sharpen() ist ziemlich schnell und arbeitet in Echtzeit. Ab einem 600er Prozessor sollte auch ein großer Bildschirm flüssig geschärft werden.
Generell gilt: Je kleiner die Fläche, desto schneller arbeitet Sharpen().
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Sharpen(%HDC,0,0,array&)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

BlackWhite (F,X,Y,B,P)

Macht aus einer Grafik auf dem Bildschirm (oder Teilen davon) ein Schwarz-Weiß Bild.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
P : Long - Priorität des Effekts, 0-255 (127=normal)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird in ein Schwarz-Weiß Bild umgewandelt. X und Y geben die linke obere Ecke an, ab wo der Bildschirm scharz-weiß gemacht wird. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Wenn sie BlackWhite() innerhalb einer Schleife mit wechselnder Priorität aufrufen, gibt das einen ganz erstaunlichen Effekt!
Je niedriger die Priorität ist, desto mehr überwiegt die Farbe Schwarz, je höher sie ist, desto mehr überwiegt die Farbe Weiß.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() /

CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
BlackWhite() ist ziemlich schnell und arbeitet in Echtzeit. Generell gilt: Je kleiner die Fläche, desto schneller arbeitet die Funktion.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion nicht verändert.

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
BlackWhite(%HDC,0,0,array&,127)
FreeFX(array&)                 'später wieder freigeben
```

Water (F,X,Y,B,H,V)

Ein toller Wasser-Effekt! Die Grafik des Bildschirms wird unter leichten Wellenbewegungen hin und her geschaukelt. Kombiniert mit vorher angewendetem FlipY() und Darken() ist diese Funktion ein Augenschmaus in allen Jump&Run-Spielen.
Wurde in Version 3.0 komplett neu geschrieben.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
H : Long - 0 oder Zeiger auf Speicher mit horizontaler Wellenbewegung
V : Long - Startzeile für vertikale Wellenbewegung
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird wie Wasser sanft hin und her geschaukelt. X und Y geben die linke obere Ecke an, ab wo der Bildschirm "nass" wird. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht. In diesem Fall werden die Arraydaten direkt geändert.
Der Aufruf von Water() sollte nicht zu schnell hintereinander erfolgen. Nur ein langsamer Ablauf wirkt realistisch, ich empfehler eine Aktualisierung nach 40-48 Millisekunden. Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Der Wassereffekt setzt sich zusammen aus einer horizontalen und vertikalen Wellenbewegung des Bildes. Wird für H eine Null angegeben, erfolgt eine fertig vorgegebene horizontale, und optimierte Welle (siehe unten). Ansonsten erwartet dieser Parameter den Zeiger auf einen Speicher mit LongInts. Es dürfen die Zahlen von -5 bis 5 verwendet werden, und als Endeerkennung ist eine 255 Pflicht. Für die Welle bedeuten diese Zahlen, dass negative Zahlen die Welle nach links, und positive Zahlen die Welle nach rechts rutschen lassen. Die vorgefertigte Welle sieht als Profancode so aus:

```
Long bereich#,0 = 0,1,1,3,1,1,3,1,1,3,2,2,2,1,3,3,3,4,3,3,3,3,4,3,3,3,3,5,3 \
                3,2,2,4,2,2,2,2,4,3,3,3,3,3,1,3,3,3,3,2,2,2,1,1,1,0,0,0,0 \

-1,-1,-1,-1,-1,-1,-1,-1,-1,-2,-2,-2,-2,-2,-2,-1,-1,-1      \
                  255
```

Die Länge der LongInt-Folge ist egal.
V bestimmt die vertikale Wellenbewegung. Hier kann der Programmierer die Wellenbewegung nicht selber bestimmen (die Gefahr hier einen Fehler zu machen wäre sehr groß). Aber er kann bestimmen, ob überhaupt eine vertikale Welle stattfinden soll, und wenn ja, ab welcher y-Position diese starten soll! Vertikale Wellen ab Position 0 lassen die Wasserbewegung oft unrealististisch aussehen Es sieht dann aus, als würde das dem Betrachter gegenüber liegende Ufer überspült werden. Für Kaimauern z.B. natürlich unerlässlich, aber eben nicht immer gewünscht. Hier ist der Programmierer aufgefordert, für jedes Bild

individuell den passenden Wert zu finden. Soll gar keine vertikale Wellenbewegung stattfinden, ist bei V einfach ein sehr hoher Wert einzutragen, z.B. einfach die Höhe des Bildes oder -1.

Die Daten im Byte-Array B werden durch Aufruf dieser Funktion nicht verändert, es sei denn F ist gleich Null!

Beispiel:

```
Long bereich#,0 = 1,2,3,4,5,4,3,2,1,0,-1,-2,-3,-4,-5,-4,-3,-2,-1,0,255

array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen

WhileLoop 256
 Water(%HDC,0,0,array&,bereich#,40)
EndWhile

FreeFX(array&)                 'später wieder freigeben
```

Rotate(F,X,Y,B1,B2,G,K)

Dreht eine Grafik um eine beliebige Achse (0-359 Grad).

```
F  : Long - Handle der Zielbitmap (HDC) oder 0
X  : Long - X Offset zum Bildschirm
Y  : Long - Y Offset zum Bildschirm
B1 : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Hintergrunds-Bild)
B2 : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten
Bytes-Arrays (Bild, das gedreht werden soll)
G  : Long - Winkel (0-359 Grad)
K  : Long - Kopiermodus
```

Ergebniss: Long - 0.

Eine gedrehte Grafik wird auf den Bildschirm gebracht (bzw. nur im Speicher gedreht). X und Y geben an, an welcher Position die gedrehte Grafik auf den Bildschirm gebracht wird. Die Grösse des sich drehenden Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde. Ecken, die dieses Rechteck verlassen, werden abgeschnitten.
Die gedrehte Grafik wird im Rechteck automatisch zentriert, was sehr praktisch ist.
Für F kann entweder eine der Profan-Bitmaps (meistens %HDC) angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Bevor Rotate() angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Für den Kopiermodus sind nur die Werte 0 und -1 zulässig. Wird als Kopiermodus die -1 gewählt, dann wird die Farbe des oberen rechten Pixels des zu drehenden Bilds als transparente Farbe gewählt, im Gegensatz zum originalen Profan-Transparentmodus wo das transparente Pixel unten links gewählt wird. Beliebige Flächen im Bild können so also durchsichtig gemacht werden.
Es werden keine Daten in den Byte-Arrays B1 und B2 durch Aufruf dieser Funktion verändert.

```
array1&=InitFX(hintergrund&)
array2&=InitFX(bild&)

' *** Bild 360 mal in Ein-Grad-Schritten rotieren lassen
'
WhileLoop 359,0,-1
 Rotate(%HDC,100,50,array1&,array2&,&loop,0)
Wend
```

```
FreeFX(array1&)
FreeFX(array2&)
```

Rotate180 (F,X,Y,B)

Dreht eine Grafik um 180 Grad.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird um 180 Grad gedreht. X und Y geben die linke obere Ecke der zu drehenden Fläche an. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert.
Rotate180() ist schneller als Rotate() und sollte bei einem Winkel von 180 Grad bevorzugt angewendet werden.

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
Rotate180(%HDC,0,0,array&)
FreeFX(array&)                 'später wieder freigeben
```

FlipX (F,X,Y,B)

Spiegelt eine Grafik in der X-Achse.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird in der X-Achse gespiegelt. X und Y geben die linke obere Ecke der zu spiegelnden Fläche an. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
FlipX(%HDC,0,0,array&)
FreeFX(array&)                 'später wieder freigeben
```

FlipY (F,X,Y,B)

Spiegelt eine Grafik in der Y-Achse.

```
F : Long - Handle der Zielbitmap (HDC) oder 0
X : Long - X Offset zum Bildschirm
Y : Long - Y Offset zum Bildschirm
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

Die Grafik des Bildschirms wird in der Y-Achse gespiegelt. X und Y geben die linke obere Ecke der zu spiegelnden Fläche an. Die Grösse des Rechtecks ergibt sich aus der Grösse der Bitmap, die in InitFX() / CreateFX() angegeben wurde.
Für F kann entweder eine der Profan-Bitmaps angegeben werden oder eine, die mit LoadFileImage() / CreateImage() erzeugt wurde. Wird für F aber 0 übergeben, dann wird die Funktion zwar ausgeführt, aber nicht auf den Bildschirm gebracht.
Bevor die Funktion angewendet werden kann, muß eine Vorbereitung mit InitFX() stattgefunden haben.
Die Daten im Byte-Array B werden durch Aufruf dieser Funktion verändert..

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
FlipY(%HDC,0,0,array&)
FreeFX(array&)                 'später wieder freigeben
```

CopyFX (F,X,Y,B)

Manuelle Kopie eines Byte-Arrays auf den Bildschirm (oder Teilen davon).

```
F : Long - Handle der Zielbitmap (HDC) erzeugte)
X : Long - X Offset in F
Y : Long - Y Offset in F
B : Long - Handle eines mit PngToFX(), InitFX() oder CreateFX() erzeugten Bytes-
Arrays (Quellbitmap)
```

Ergebniss: Long - 0.

```
array&=InitFX(exhdc&)          'Handle (Byte-Array) zu einer Bitmap erzeugen
FlipX(%HDC,0,0,array&)
FreeFX(array&)                 'später wieder freigeben
```

bewirkt das gleiche wie...

```
array&=InitFX(exhdc&)
FlipX(0,0,0,array&)
CopyFX(%HDC,0,0,array&)
FreeFX(array&)
```

DSoundInit (F)

Initialisiert DirectSound, ein Teil von DirectX. Um also die DSound-Funktionen der ProSpeed.dll zu nutzen, muß DirectX installiert sein.

DSoundInit() muß vor Verwendung aller anderen DSound-Funktionen aufgerufen werden.

```
F : Long - Handle des Profan-Fensters (%hwnd)
```

Ergebniss: Long - 0 = Ok / ungleich 0 = Fehler.

Wenn DSoundInit() aufgerufen wird, muß schon ein Fenster (F) geöffnet sein.

```
DSoundInit(%hwnd)
```

DSoundDeInit ()

Deinitialisiert DirectSound wieder und gibt den Speicher frei. Nach dieser Funktion darf keine DSound-Funktion mehr aufgerufen werden.

Ergebniss: Long - 0

DSoundLoad (S)

Lädt eine Wave-Datei zum späteren Abspielen.

```
S : Long - Zeiger eines Strings mit dem Namen einer Wave-Datei
```

Ergebniss: Long - 0 = Fehler / ansonsten Zeiger auf ein Buffer-Array (wichtig für DSoundUnLoad()).

Eine Wav-Datei wird geladen und in einem Buffer-Array gespeichert.
Eigentlich werden in dem Buffer-Array 15 Kopien des Sound angelegt, plus eine Zählervariable. Diese 15 Kopien belegen intern aber denselben DirectSound-Speicher, sodaß kein Speicher vergeudet wird.
Warum 15 Kopien? Jede Kopie kann nur einmal zur gleichen Zeit abgespielt werden. Wenn wir aber 15 Kopien haben, kann auch 15 mal ein und dergleiche Sound gleichzeitig abgespielt werden, und das sollte für jedes Spiel reichen, was wir entwerfen...

```
text$="Test.wav"
buffer&=DSoundLoad(addr(text$))
```

DSoundUnLoad (A)

Diese Funktion gibt ein Buffer-Array wieder frei, das mit DSoundLoad() erstellt wurde.

```
A : Long - Zeiger auf ein Buffer-Array, das mit DSoundLoad() erstellt wurde
```

Ergebniss: Long - 0

Die Kopien eines Sounds und der Sound selber werden wieder freigegeben.

```
DSoundUnLoad(buffer&)
```

DSoundGetNextBuffer (A)

Diese Funktion ermittelt einen einzelnen Buffer aus einem Buffer-Array.

```
A : Long - Zeiger auf ein Buffer-Array, das mit DSoundLoad() erstellt wurde
```

Ergebniss: Long - Zeiger auf den Buffer, der als nächstes mit DSoundPlay() gespielt wird

Das Ergebniss dieser Funktion ist einer der 15 Buffer aus einem Buffer-Array, und zwar der Buffer, der als nächstes mit DSoundPlay() gespielt wird. Mit diesem Zeiger kann die Lautstärke oder das Panning eines Sounds verändert werden, bevor der Sound abgespielt wird.

Bitte beachten, das playbuffer& ein Zeiger auf einen Speicherbereich ist, während buffer& ein Zeiger auf ein Array mit Speicherbereichen ist...

```
playbuffer&=DSoundGetNextBuffer(buffer&)
```

DSoundPlay (A,L)

Diese Funktion spielt einen Sound ab, der mit DSoundLoad() geladen wurde.

```
A : Long - Zeiger auf ein Buffer-Array, das mit DSoundLoad() erstellt wurde
L : Long - Flag für Loop (abspielen in einer Schleife). 0 = kein Loop / 1 = Loop
```

Ergebniss: Long - Zeiger auf den spielenden Buffer.

Das Ergebniss dieser Funktion ist einer der 15 Buffer aus einem Buffer-Array, und zwar der Buffer, der gerade gespielt werden soll. Dieser Wert kann auch durch DSoundGetNextBuffer() abgefragt werden, ohne den Sound abzuspielen.

Bitte beachten, das playbuffer& ein Zeiger auf einen Speicherbereich ist, während buffer& ein Zeiger auf ein Array mit Speicherbereichen ist...

```
playbuffer&=DSoundPlay(buffer&,0)
```

DSoundStop (B)

Diese Funktion stoppt einen Sound, der gerade gespielt wird.

```
B : Long - Zeiger auf einen Buffer, der von DSoundPlay() oder
DSoundGetNextBuffer() übergeben wurde.
```

Ergebniss: Long - 0

```
DSoundStop(playbuffer&)
```

DSoundSetVolume (B,V)

Diese Funktion ändert die Lautstärke eines Sounds.

```
B : Long - Zeiger auf einen Buffer, der von DSoundPlay() oder
DSoundGetNextBuffer() übergeben wurde.
V : Long - Lautstärke, -10000 = ganz leise / 0 = ganz laut.
```

Ergebniss: Long - 0

Diese Funktion ändert die Lautstärke für den Sound im Buffer B. Die Werte für V reichen von -10000 bis 0.

```
DSoundSetVolume(playbuffer&,-575)
```

DSoundSetPan (B,P)

Diese Funktion ändert das Panning eines Sounds.

```
B : Long - Zeiger auf einen Buffer, der von DSoundPlay() oder
DSoundGetNextBuffer() übergeben wurde.
P : Long - Panning, -10000 = ganz links / 10000 = ganz rechts.
```

Ergebniss: Long - 0

Diese Funktion ändert das Panning für den Sound im Buffer B. Die Werte für P reichen von -10000 (ganz links) bis 10000 (ganz rechts). Ein Wert von 0 steht für eine gleichmäßige Verteilung (Mitte).

```
DSoundSetPan(playbuffer&,5000)
```

DSoundSetGlobalVolume (V)

Diese Funktion ändert die Lautstärke aller Sounds gleichzeitig.

```
V : Long - Lautstärke, -10000 = ganz leise / 0 = ganz laut.
```

Ergebniss: Long - 0

```
DSoundStop(-575)
```

DSoundSetGlobalPan (P)

Diese Funktion ändert das Panning aller Sounds gleichzeitig.

```
P : Long - Panning, -10000 = ganz links / 10000 = ganz rechts.
```

Ergebniss: Long - 0

```
DSoundStop(5000)
```

DSoundGetInterfaces (S)

Diese Funktion macht ProSpeed's DirectSound Funktionen kompatibel zu Sebastian Königs DirectSound Profan-Prozeduren.

```
S : Long - Zeiger auf eine Variable, die den Schnittstellen-Zeiger des Primary-
Buffers aufnimmt.
```

Ergebniss: Long - Schnittstellen-Zeiger des IDirectSound Objektes.

Sebastian Königs Prozedur DSInitFromProSpeed() benutzt DSoundGetInterfaces() um Dll und Profan-Vorlage kompatibel zu machen. So können sie beides kombinieren. Nur für Profi's, die ProSpeed und

Sebastians Vorlagen kombinieren wollen.

Beispiel:

```
IDirectSound&=DSoundGetInterfaces(Addr(DSBPrimary&))
```

Stars (Z,R,L,O,R,U,A,G,F)

Erzeugt ein bewegtes Sternenfeld, bei dem die Sterne dem Betrachter entgegen kommen. Stars() arbeitet automatisch in einem eigenen Thread, bis die Funktion NoStars() das Sternenfeld beendet.

```
Z : Long - Handle des HDC's, in den die Sterne gezeichnet werden, meistens wohl
%HDC
R : Long - 0, oder Handle des HDC's, der für die Restaurierung der Sternenpunkte
genommen wird
L : Long - linke Begrenzung des Sternenfelds (relativ zum Fenster)
O : Long - obere Begrenzung des Sternenfelds (relativ zum Fenster)
R : Long - rechte Begrenzung des Sternenfelds (relativ zum Fenster)
U : Long - untere Begrenzung des Sternenfelds (relativ zum Fenster)
A : Long - Anzahl Sterne
G : Long - Geschwindigkeit der Sterne (niedrig = schnell)
F : Long - Flags
```

Ergebniss: Long - Bei Fehler 0, ansonsten Adresse des Threads.

A Sterne kommen auf den Betrachter zugeflogen. Die Sterne kommen aus der Mitte des in L,O,R und U angegebenen Rechtecks und wandern dann zum Rand. Mit der Funktion OptionStars() kann aber noch nachträglich der Mittelpunkt des Sternenfelds und die Richtung/Geschwindigkeit der Sterne verändert werden.
Es kann immer nur ein Sternenfeld gleichzeitig erzeugt werden.
Die Größe des Feldes, das mit L, O, R und U bestimmt wird, muß mindestens die Größe 320 x 320 besitzen, kleinere Sternenfelder sind nicht gestattet.
Wenn Sterne zusammen mit Sprites programmiert werden, dann sollte für R der Wert von Kopie 2 (K2) angegeben werden (siehe InitSprites).
Wird für R eine Null angegeben, dann wird der Bildschirm nur mit schwarzen Pixeln restauriert.
Die Anzahl Sterne A sollte bei 100 bis 300 liegen, höhere Werte strukturieren ansonsten den Bildschirm zu sehr.
Für G sind Werte von 2 bis 256 sinnvoll, wobei sich die Sterne beim Wert 2 am schnellsten bewegen. Der optimale Wert liegt hier bei 12.
Für F stehen diese Flags momentan zur Auswahl:

```
0 = Normaler Modus.
1 = Nur diejenigen Pixel restaurieren, deren Hintergrundsfarbe schwarz
(RGB(0,0,0)) ist.
   Sinnvoll nur im Zusammenhang mit einem HDC in R. Die Sterne werden nur dort
gezeichnet, wo
   eine schwarze Fläche ist, sonstige Grafiken verdecken quasi die Sterne.
```

Beispiel:
Eine Demo zu den Stars finden sie im Sourcecode-Ordner in der Datei **FA_Stars.prf**

```
Stars(%HDC,0,0,0,800,600,200,64,0)
```

OptionStars (XO,YO,X,Y,G)

Diese Funktion beeinflußt das Verhalten der Sterne in einem Sternenfeld, das mit Stars() generiert wurde.

```
XO : Long - X-Offset, diesen Wert zum Mittelpunkt eines Sternenfelds addieren
YO : Long - Y-Offset, diesen Wert zum Mittelpunkt eines Sternenfelds addieren
X  : Long - X-Geschwindigkeit der Sterne um diesen Wert addieren
Y  : Long - Y-Geschwindigkeit der Sterne um diesen Wert addieren
G  : Long - Geschwindigkeit der Sterne (siehe G bei Stars())
```

Ergebniss: Long - 0

Ein Sternenfeld kann hiermit beeinflußt werden.
Mit XO und YO kann der Mittelpunkt des Sternenfelds verlegt werden, die Sterne tauchen dann also nicht mehr unbedingt in der Mitte des Feldes auf. Negative Werte verlegen den Mittelpunkt nach links, bzw. nach oben. Positive Werte nach rechts, bzw. nach unten.
X und Y addieren den Wert zu der normalen Geschwindigkeit jedes Sterns. Den Sternen kann also eine Tendenz in eine bestimmte Richtung gegeben werden, wobei negative Werte Tendenzen nach links, bzw. oben erzeugen. Hier sind Werte von -10 bis 10 sinnvoll.
G ändert die Bewegungs-Geschwindigkeit aller Sterne.
OptionStars() kann beliebig oft aufgerufen werden, wobei die neuen Werte die vorher gesetzten Werte überschreiben.

```
OptionStars(100,100,2,2,4)
```

StatusStars ()

Diese Funktion prüft, ob ein Sternenfeld noch läuft, oder schon beendet wurde.

Ergebniss: Long - Status des Sternenfelds

Für Ergebniss gibt es diese beiden Möglichkeiten:

```
0 = Sternenfeld ist noch aktiv, oder es wurde gar keins erzeugt.
1 = Sternenfeld wurde beendet.
```

Diese Funktion ist sehr sinnvoll, wenn ein Sternenfeld via NoStars() mit Parameter 3 beendet wurde. Bei Parameter 3 werden alle Sterne noch weiterbewegt, bis sie aus dem Blickfeld verschwunden sind, erst dann wird das Sternenfeld beendet.

NoStars (P)

Diese Funktion beendet ein Sternenfeld. Hierbei gibt es mehrer Methoden.

```
P : Long - Parameter für das Beenden des Sternenfelds
```

Ergebniss: Long - 0

Ein Sternenfeld wird, mit einer von mehreren Methoden, beendet. Für P kann einer der folgenden Werte eingesetzt werden:

```
0 = Sternenfeld nicht abbrechen.
1 = Sternenfeld wird sofort beendet, ohne die Sterne vom Bildschirm zu löschen.
2 = Sternenfeld wird sofort beendet, alle Sterne werden aber noch vom Bildschirm
gelöscht (Bild
    wird restauriert).
```

```
3 = Sternenfeld wird langsam beendet. Es werden keine neuen Sterne mehr erzeugt,
aber die
   vorhandenen Sterne werden noch weiter bewegt, bis sie aus dem Blickfeld
verschwunden sind.
   Ob das Sternenfeld schon beendet ist, kann mit StatusStars() abgefragt
werden.

NoStars(3)
```

InitPixelEffects()

Initiert die Pixeleffekte.

Ergebniss: Long - 0 = Fehler / größer 0 = Initierung erfolgreich

Wichtig: Bevor SetPixelEffect() benutzt werden kann, muß mit InitPixelEffects() der Pixel-Thread vorbereit werden.
DeInitPixelEffects() beendet den Thread schließlich wieder.

SetPixelEffect(Z,R,X,Y,P,S,H,F,A,M,L)

Diese Funktion startet einen Pixeleffekt.

```
Z : Long - Handle des HDC's, in dem die Pixel gezeichnet werden sollen, meistens
wohl %HDC
R : Long - 0, oder Handle des HDC's, der für die Restaurierung der Pixel
genommen wird, z.B. %HDC2
X : Long - X-Position des Effekts
Y : Long - Y-Position des Effekts
P : Long - Parameter für Farbmodus (M)
S : Long - Startstreuung der Pixel (1 - 160)
H : Long - Heftigkeit des Pixelausstoßes (3 - 31)
F : Long - Schwerkraft-Richtung (Fallrichtung der Pixel)
A : Long - Anzahl Pixel (3 - 1000)
M : Long - Farbmodus der Pixel (nicht Farbe!)
L : Long - Lebensdauer der Pixel (10 - 300)
```

Ergebniss: Long - 0

Es wird ein Pixeleffekt (vorwiegend wohl eine Explosion) gestartet. Vorher müßen aber die Pixeleffekte mit InitPixelEffects() initiert werden (nur einmal, nicht für jeden Pixeleffekt!).
Z ist der Ausgabe-HDC (Fenster), meistens wohl %HDC.
R ist der Restaurierungs-HDC (z.B. %HDC2 oder ein mit ProSpeed erzeugter HDC). Wenn R = 0 ist, dann werden alle gesetzten Pixel einfach mit Schwarz restauriert.
X und Y geben an, wo im HDC Z der Effekt positioniert wird. Hierbei wird eine gewisse Startstreuung S vorgenommen, damit nicht alle Pixel direkt an Position XY beginnen.
Die Heftigkeit eines Effekts wird mit H eingestellt, je höher der Wert, desto schneller bewegen sich die Pixel.
Nach einer gewissen Zeit beginnt eine Schwerkraft die Pixel zu beeinflussen. Hierbei kann mit F die Richtung der Schwerkraft eingestellt werden:

```
0 = keine Schwerkraft.
1 = Schwerkraft läßt die Pixel nach rechts hin "fallen".
2 = Schwerkraft läßt die Pixel nach unten hin "fallen".
3 = Schwerkraft läßt die Pixel nach links hin "fallen".
4 = Schwerkraft läßt die Pixel nach oben hin "fallen".
```

Die Anzahl Pixel wird bei A angegeben. Hierbei erscheinen nicht alle Pixel gleichzeitig, sondern mit einer kleinen Verzögerung.
Die Lebensdauer der Pixel wird mit L eingestellt, nach dieser Zeit erlöschen sie.
Der Farbmodus inm Pixeleffekt wird unter M eingestellt. Bisher gibt es drei Möglichkeiten:

```
0 = Die Anfangs-Farbe der Pixel wird bei P eingestellt. Dann werden die Pixel
immer dunkler, bis sie
    ganz schwarz sind (Abdunklungs-Modus).
1 = Die Farbe der Pixel wird bei P eingestellt (Einfarb-Modus).
2 = Die Pixel blinken in zufälligen Farben (Zufallsmodus).
```

Ein Pixeleffekt ist komplett beendet, wenn die Lebensdauer aller Pixel abgelaufen ist. Der fertige Pixeleffekt wird dann nicht mehr weiter im Thread bearbeitet.
Momentan dürfen 256 verschiedene Pixeleffekte gleichzeitig ablaufen (sofern genug Prozessorzeit zur Verfügung steht) !
Je nach Angaben können strukturierte oder zufällige Pixelexplosionen erzeugt werden, großflächige oder kleine, heftige oder langsame. Spielen sie ruhig mal mit den Werten.
Ein schönes Anwendungsbeispiel sind explodierende Sprites, das Sprite wird gelöscht und an seiner Position eine Explosion gestartet, sieht sehr professionell aus!

```
InitPixelEffects()
SetPixelEffect(%HDC,0,200,200,$FFe020,24,13,2,250,0,120)
...
DeInitPixelEffects()
```

DeInitPixelEffects()

Beendet die Pixeleffekte.

Ergebniss: Long - 0

DeInitPixelEffects() beendet den Thread wieder, der mit InitPixelEffects() gestartet wurde.
Muß vor dem Programmende erfolgen, um allen Speicher wieder freizugeben.

FindPath (B,BA,W,X1,Y1,X2,Y2,F)

Diese Funktion findet einen Weg in einer Pixelgrafik (Bitmap). Dank künstlicher Intelligenz wird ein relativ kurzer Weg gefunden, wobei schwarze Flächen in der Bitmap Hindernisse und weiße Flächen begehbare Stellen kennzeichnen.

```
B  : Long - Zeiger auf einen Speicherbereich (Bereichsvariable), der später die
Wegdaten aufnimmt
BA : Long - Zeiger auf ein Byte-Array (einer Bitmap), das mit InitFX() erzeugt
wurde
W  : Long - Wegbreite, es wird ein Weg gesucht, der mindesten W Pixel breit /
hoch ist (1 - ...)
X1 : Long - X-Position innerhalb der Bitmap, von dem die Suche ausgeht
(Startpunkt)
Y1 : Long - Y-Position innerhalb der Bitmap, von dem die Suche ausgeht
(Startpunkt)
X2 : Long - X-Position innerhalb der Bitmap, von bei der die Suche endet
(Zielpunkt)
Y2 : Long - Y-Position innerhalb der Bitmap, von bei der die Suche endet
(Zielpunkt)
```

```
F  : Long - Flags
```

Ergebnis: Long

```
-1 = Ein Fehler ist aufgetreten (Speichermangel)
0  = Kein Weg gefunden
>0 = Anzahl Bytes an Wegdaten, die in B generiert wurden. Bei einem Wert über
Null war die Suche war also      erfolgreich.
```

Eine neue aufwendige Funktion, um z.B. Sprites durch eine labyrinth-artige Landschaft zu führen
FindPath() arbeitet sehr schnell und findet auch komplizierte Wege oftmals innerhalb einer Millisekunde.
Zuersteinmal benötigt die Funktion eine Maskenbitmap mit den vorhanden Wegen und Hindernissen. Vom Hintergrund muß also ein scharz-weiß Bild vorhanden sein, worin schwarze Flächen für Hindernisse (Mauern usw.) stehen und weiße Flächen für Wege (freie Plätze).
Diese Maskenbitmap muß dann mit InitFX() in ein Byte-Array (BA) umgewandelt werden. Mit so einem Byte-Array kann FindPath sehr schnell arbeiten. Und immer daran denken, das die Breite so einer Byte-Array-Grafik immer glatt durch vier teilbar sein muß...

Wenn ein Weg gefunden wurde, schreibt FindPath() die Wegdaten in den Speicherbereich B.
Wegdaten sind immer Integer-Werte, für jeden Koordinaten-Punkt schreibt die Funktion einen X-Wert und einen Y-Wert, pro Wegpunkt also 2 Integer (=4 Bytes). Wie viele Wegpunkte (Koordinaten-Paare) nötig sind, um von Startpunkt zum Zielpunkt zu gelangen, liegt an der Länge des gefundenen Weges und an der Wegbreite (je breiter der Weg, desto weiter liegen die Wegpunkte auseinander, desto weniger Wegpunkte werden benötigt).
B muß groß genug dimensioniert sein, um alle Wegpunkte aufnehmen zu können. Leider läßt sich vorher nicht berechnen, wieviele Bytes benötigt werden. Nach folgender Formel kann die maximale Byteanzahl berechnet werden, in der Praxis kann sie aber nie erreicht werden:
((Breite der Bitmap * W) * (Höhe der Bitmap * W)) *64
Ich persönlich nehme hierfür einen Speicher mit 200.000 Bytes, das reicht für jede Wegberechnung.
W gibt an, wie breit (in Pixel) ein Weg mindestens sein muß, damit er gefunden wird. Machen sie Wege aber immer breiter, das ist auf jeden Fall sicherer (besondern in Kurven). Außerdem können sie in F ein Flag setzten, um Hindernisse / Mauern vor der Wegberechnung künstlich breiter zu machen, was die Wege natürlich automatisch schmaler macht.
Sinnvolle Werte für W gehen von 6 bis 48. Werte von 1-5 sind zu klein und die Ausführung von FindPath() dauert entsprechend länger, Werte größer als 48 sind zu grob und die Bitmap wird intern zu sehr aufgepixelt. Bei meinen Tests brachte ein Wert von 12 für W die besten Ergebnisse. Soll ein Sprite durch den Weg geschickt werden, dann ist der beste Wert:
(Breite des Sprites + Höhe des Sprites) / 2.
X1 und Y1 bezeichnen den Startpunkt in der Bitmap, X2 und Y2 den Zielpunkt der Wegsuche.
F steht für die Flags, bisher gibt es nur zwei Flags:

```
0 = Normaler Modus
1 = Hindernisse / Mauern vor der Wegsuche künstlich vergrößern
```

Flag 1 wurde deswegen eingeführt, weil bei der Wegsuche der Weg oft haarscharf an Hindernissen vorbei führt. Wenn ein Sprite diesen Weg abläuft, sieht es dann so aus, als kollidiere das Sprite mit den Hindernissen. Flag 1 verbreitert unsichtbar die Hindernisse (das Bild in BA wird nicht verändert!) und verhindert diesen Effekt. Natürlich werden dadurch Wege / freie Flächen etwas schmaler, sorgen sie also für genügend breite Wege.
Nachdem ein Weg berechnet wurde, sollte er noch mit SmoothPath() geglättet werden.

Beispiel:

Der Beispielcode **FA_Wegsuche.prf** zeigt sehr schön, wie man eine Wegsuche programmiert.

```
x&=FindPath(bereich#,array&,16,200,200,400,400,0)
```

SmoothPath (B,A,O)

Diese Funktion glättet Wege, die mit FindPath() oder PixelToMemory() errechnet wurden oder die mit dem Programm "Pathfinder" von Moritz Kersten (liegt ProSpeed als Shareware bei) erstellt wurden. Kurven werden runder und Kanten werden geglättet.

```
B : Long - Zeiger auf einen Speicherbereich (Bereichsvariable), der die Wegdaten
enthält
A : Long - Anzahl Bytes an Wegdaten in B
O : Long - Gibt an, wie oft der Pfad geglättet werden soll
```

Ergebniss: Long - 0

Die Wegdaten, die In B stehen, werden geglättet.
A gibt an, wieviele Bytes an Wegdaten in B stehen. Pro Koordinaten-Paar sind das ein X-Achsen-Wert und ein Y Achsen-Wert (Integer), also 2 Integer = 4 bytes.
O gib an, wie oft die Glättung vorgenommen wird, bei 1 wäre das einmal. Die besten Glättungs-Ergebnisse bringen Werte von 2-4. Je öfter geglättet wird, desto mehr werden die ursprünglichen Wegdaten verfälscht.

```
SmoothPath(bereich#,64,4)
```