Inhaltsverzeichnis

Index der benutzten API-Funktionen

Quellenangabe:
Visual Basic Programmer's Guide to the Windows API,Daniel Appleman, Copyright (c) 1993 by Ziff-Davis Press

Programmverknüpfung

Im Dateimanager können Sie sogenannte Verknüpfungen vornehmen. Dabei bestimmen Sie, welches Programm gestartet werden soll und die angegebene Datei laden soll, wenn die entsprechende Datei geöffnet wird. Die Verknüpfung erfolgt jeweils über die Dateiendung. Bei einem Doppelklick auf eine Datei mit der Endung .TXT wird standardmässig das Programm NOTEPAD.EXE gestartet und die angeklickte Datei geladen. Windows bietet nun ein API, um anhand eines Dateinamens den Programmnamen des verknüpften ausführbaren Programmes ausfindig zu machen. Über die folgende Funktion können Sie den zugehörigen Dateinamen in Erfahrung bringen:

```
Declare Function FindExecutable% Lib "shell.dll" (ByVal lpszFile$, ByVal lpszDir$,
ByVal lpszResult$)Function GetLinkedAppName (pfad$)  Dim i%  Dim Resultat$
Resultat$ = String$(256, " ")  i% = FindExecutable%(pfad$, "", Resultat$)
GetLinkedAppName = Left$(Resultat$, InStr(Resultat$ + Chr$(0), Chr$(0)) - 1)End
Function
```

Damit sind Sie in der Lage, die Dateimanager-Funktionalität nachzubilden. Erzeugen Sie eine DateiListBox und fügen Sie im Click-Ereignis folgenden Code ein:

```
Sub File1_DblClick ()  Dim tmp$, i%  tmp$ = File1.Path  If Right$(tmp$, 1) <> "\"
Then tmp$ = tmp$ + "\"  On Error Resume Next  i% = Shell(Trim$
(GetLinkedAppName(tmp$ + File1) + " " + tmp$ + File1), 1)  If Err Then    MsgBox
"Diese Datei ist mit keiner Anwendung verknüpft", 48, "Fehler"    Exit Sub  End
IfEnd Sub
```

Fensterhandling

Fenster einfrieren. Manchmal kann es wünschenswert sein, den Aufbau eines Fensters zu verstecken. Werden z.B. sehr viele Steuerelemente auf einer Form neu positioniert, so kann man den Aufbau mitverfolgen. Ausserdem dauert es recht lange. Frieren Sie das Fenster vor dem Aufbau ein und geben es erst nach dem Aufbau wieder frei, so gewinnen Sie Zeit und erhalten erst noch ein schöneres Finish. Mit dieser Funktion kann auch das Füllen von Listboxen beschleunigt werden.

```
Declare Function LockWindowUpdate Lib "User" (ByVal hWnd As Integer) As
IntegerFunction Freeze(hWnd as Integer)  Dim Ret%  Ret% =
LockWindowUpdate(hWnd%)End FunctionFunction UnFreeze(hWnd%)  Dim Ret%   Ret% =
LockWindowUpdate(0)End Function
```

Nach dem Aufruf von MeltWindow sollten Sie dafür sorgen, dass alle betroffenen Fenster neu gezeichnet werden. Um z.B. die aktive Form einzufrieren, benutzen Sie den Aufruf i% = Freeze((Me.hWnd)). Die doppelten runden Klammern werden benötigt für die Typumwandlung. Um eine Listbox einzufrieren, benutzen Sie den Aufruf i% = Freeze ((List1. hWnd)).

Fenstertitel des aktiven Fensters ermitteln
Mittels folgender Funktion können Sie den Fenstertitel des gerade aktiven Fensters ermitteln. Dies ist besonders dann interessant, wenn es sich nicht um ein eigenes Fenster handelt. Mit der Routine AppActivate können Sie das Fenster zu einem späteren Zeitpunkt wieder aktivieren.

```
Declare Function GetActiveWindow Lib "User" () As IntegerDeclare Function
GetWindowText Lib "User" (ByVal hWnd As Integer, ByVal lpString As String, ByVal
aint As Integer) As IntegerFunction AktFensterTitel$ ()  Dim Titel$, laenge%
Titel$ = Space$(255)  laenge% = GetWindowText(GetActiveWindow(), Titel$, 255)
AktFensterTitel$ = Left$(Titel$, laenge%)End Function
```

Schwebendes Fenster
Manchmal möchte man verhindern, dass ein Fenster durch andere Fenster verdeckt werden kann. Diesen Effekt kann man mittels eines API-Aufrufs erzielen. Die folgende Routine schaltet das aktuelle Fenster (Me) um, so dass es immer im Vordergrund bleibt. Ein erneuter Aufruf der Routine schaltet das aktuelle Fenster wieder in den Normalzustand um.

```
Declare Sub SetWindowPos Lib "User" (Byval hWnd as integer, Byval hWndInsertAfter
as Integer, Byval X as Integer, Byval Y as Integer, Byval cx as Integer, Byval cy
as Integer, Byval wFlags as Integer)
Sub SwitchOnTopMode ()  Static Modus%  If Not Modus% Then    SetWindowPos
form1.hWnd, -1, 0, 0, 0, 0, &H50 'Immer sichtbar  Else    SetWindowPos form1.hWnd,
-2, 0, 0, 0, 0, &H50 'Normalzustand  End If  Modus% = Not Modus%End Sub
```

Mauszeiger auf Bereich beschränken
Mittels API-Funktionen kann man den Bereich, der mit dem Mauszeiger erreichbar sein soll, einschränken. Wollen Sie z.B. erreichen, dass der Anwender nicht mit dem Mauszeiger aus einem Bildfeld herausfahren kann, können Sie dies folgendermassen tun.

```
Type RECT    left As Integer    top As Integer    right As Integer    bottom As
IntegerEnd TypeDeclare Sub ClipCursorRect Lib "User" Alias "ClipCursor" (lpRect As
RECT)Declare Sub ClipCursorClear Lib "User" Alias "ClipCursor" (ByVal
lpRect&)Declare Sub GetWindowRect Lib "User" (ByVal hWnd%, lpRect As RECT)Sub
LimitCursor (hWnd%)  Dim r As RECT  GetWindowRect hWnd%, r  ClipCursorRect rEnd
SubSub FreeCursor ()  ClipCursorClear 0End Sub
```

Um nun den Mauszeiger-Aktionsbereich festzulegen übergeben Sie der Funktion LimitCursor den Fensterhandle des entsprechenden Bereichs. Bitte beachten Sie unbedingt, dass die Prozedur FreeCursor spätestens beim Verlassen des Programmes aufgerufen wird, denn die Einstellung bleibt sonst auch nach demerlassen des Beispiel, um den Cursor auf die aktuelle Form zu beschränken:

```
LimitCursor((me.hwnd))
```

Beispiel, um den Cursor auf das Bildfeld Picture1 zu beschränken:

```
Limitcursor((picture1.hwnd))
```

Sie können den Bereich, auf den Sie den Mauszeiger beschränken möchten, auch “von Hand” bestimmen, indem Sie in der Funktion LimitCursor statt die Funktion GetWindowRect aufzurufen direkt die entsprechenden Werte der Struktur RECT setzen. Dabei müssen Sie aber beachten, dass das API die X/Y-Koordinaten in Pixeln und relativ zum gesamten Bildschirm erwartet.

Mauszeigerposition bestimmen
Solange der Mauszeiger sich über einem Formular der eigenen Visual Basic Anwendung befindet, kann man anhand des MouseMove-Ereignisses jeweils herausfinden, wo sich der Mauszeiger gerade befindet. Wird der Mauszeiger jedoch über ein fremdes Programm oder über den Desktop bewegt, so kann man mit Visual Basic die aktuelle Position nicht mehr herausfinden. Das Windows-API GetCursorPos hilft bei der Lösung des Problems. Die folgende Prozedur liefert die X/Y-Position des Mauszeigers, egal, wo er sich gerade befindet. Da die Windows-APIs immer in Pixeln arbeiten, können Sie beim Aufruf der Routine über das Flag Twips% angeben, ob die Koordinaten in Twips (Twips% = True) oder in Pixeln (Twips% = False) zurückgegeben werden sollen.

```
Type POINTAPI    X As Integer    Y As IntegerEnd Type
```

```
Declare Sub GetCursorPos Lib "User" (lpPoint As POINTAPI)Sub GetMousePos (X%, Y%,
Twips%)  Dim p As POINTAPI  GetCursorPos p  If Twips% Then    X% = p.X *
screen.TwipsPerPixelX    Y% = p.Y * screen.TwipsPerPixelY  Else    X% = p.X    Y% =
p.Y  End IfEnd Sub
```

Mauszeiger auf bestimmte Position setzen

Mit dem Windows-API SetCursorPos kann der Mauszeiger per Programm an einen bestimmten Ort auf dem Bildschirm gesetzt werden. Wie bei API-Routinen üblich, werden von SetCursorPos die X/Y-Koordinaten in Pixeln und nicht in Twips erwartet. Die Position muss immer auf den ganzen Bildschirm bezogen angegeben werden. Die folgende Funktion ermöglicht nun die Positionierung, wobei zwischen Twix und Pixeln umgeschaltet werden kann wie bei der Funktion GetMousePos.

```
Declare Sub SetCursorPos Lib "User" (ByVal X As Integer, ByVal Y As Integer)Sub
SetMousePos (X%, Y%, Twips%)  Dim p As POINTAPI  If Twips% Then    p.X = X% /
screen.TwipsPerPixelX    p.Y = Y% / screen.TwipsPerPixelY  Else    p.X = X%    p.Y
= Y%  End If  SetCursorPos p.X, p.YEnd Sub
```

Warten vor dem Weiterfahren

In der Multitaskingumgebung von Windows wird es direkt schwierig, ein Programm dazu anzuhalten, eine bestimmte Zeit lang einfach zu warten, bevor es weiter abgearbeitet wird. Mit der folgenden Routine können Sie Ihr Programm dazu veranlassen, eine bestimmte Zeit in Millisekunden zu warten. Dazu wird das API GetTickCount benutzt, das die Anzahl Millisekunden liefert, seit der die aktuelle Windows-Sitzung läuft. Über einen Zähler wird die Möglichkeit einer Endlosschleife vermieden. Diese Massnahme schränkt jedoch die maximale Wartezeit ein.

```
Declare Function GetTickCount Lib "User" () As LongSub Wait (Millisekunden%)  Dim
zaehler&  Dim StartZeit As Long, EndZeit As Long  StartZeit = GetTickCount()  Do
zaehler& = zaehler& + 1    If (GetTickCount() - StartZeit) > Millisekunden% Then
Exit Do  Loop Until zaehler& = 60000End Sub
```

Prüfen auf DOS-Anwendung

Mit der folgenden Funktion können Sie ein Fenster daraufhin untersuchen, ob es eine DOS-Anwendung enthält oder nicht.

```
Declare Function GetWindowTask Lib "User" (ByVal hWnd As Integer) As IntegerDeclare
Function IsWinOldApTask Lib "Kernel" (ByVal hTask As Integer) As IntegerFunction
IsDOS (hWnd As Integer) As Integer  IsDOS = IsWinOldApTask(GetWindowTask(hWnd))End
Function
```

Mehrfachstart einer Anwendung unterbinden

Oft möchte man verhindern, dass die eigene Anwendung mehrfach gestartet werden kann. Dies selbst ist eigentlich kein Problem; schon etwas kniffliger wird es, wenn man die Erst-Instanz des Programmes aktivieren möchte, bevor man die zusätzlich gestartete Anwendung wieder verlässt. Folgende Routine löst das Problem. Die Routine sollte entweder in der Startroutine Sub Main oder im Load-Ereignis der Startform aufgerufen werden. Das benutzte Windows-API gehört zu den nicht dokumentierten APIs.

```
Declare Sub SwitchToThisWindow Lib "user" (ByVal hWnd%, ByVal StateNormal%)Sub
AppActivatePreviousInstance ()  Dim ThisAppTitle$  If App.PrevInstance Then
ThisAppTitle$ = App.Title    App.Title = ""    On Error Resume Next    AppActivate
ThisAppTitle$    SwitchToThisWindow GetActiveWindow(), True    End  End IfEnd Sub
```

Feststellen, wie ein Steuerelement den Focus erhalten hat

Um festzustellen, ob ein Steuerelement den Focus durch die Betätigung der TAB-Taste, durch eine ALT-Tastenkombination (Kurztaste), einen Mausklick oder eine Programmanweisung erhalten hat, kann direkt im GotFocus-Ereignis des entsprechenden Steuerelementes folgender Code verwendet werden:

```
Declare Function GetAsyncKeyState Lib "User" (ByVal vKey As Integer) As
IntegerGlobal Const ON_TAB = 1Global Const ON_ALT = 2Global Const ON_MOUSE =
3Global Const ON_ELSE = 4Function GotFocusON ()  Const KEY_TAB = &H9  Const
KEY_MENU = &H12  Const KEY_LBUTTON = &H1  If GetAsyncKeyState(KEY_TAB) = &H8001
Then    GotFocusON = ON_TAB  ElseIf GetAsyncKeyState(KEY_MENU) = &H8001 Then
GotFocusON = ON_ALT  ElseIf GetAsyncKeyState(KEY_LBUTTON) = &H8001 Then
GotFocusON = ON_MOUSE  Else    GotFocusON = ON_ELSE  End IfEnd Function
```

Tip: Diese Funktion kann Ihnen gute Dienste leisten, wenn Sie nach dem allgemein verwendeten Muster von Windows arbeiten möchten und beim Anspringen eines Textfeldes mittels TAB den Inhalt des Textfeldes markieren möchten. Dies können Sie innerhalb des GotFocus-Ereignisses des entsprechenden Textfeldes folgendermassen erreichen (hier am Beispiel des Textfeldes Text1):

```
Sub Text1_GotFocus ()  If GotFocusON() <> ON_MOUSE Then    Text1.SelStart = 0
Text1.SelLength = Len(Text1.Text)  End IfEnd Sub
```

Selektion in einem Kombinationsfeld

Vor allem in Datenbankanwendungen muss man oft ein Kombinationsfeld oder ein Listenfeld nach einem bestimmten Eintrag durchsuchen und diesen zum aktuellen Eintrag machen. Dies lässt sich zwar gut mit Visual Basic selbst lösen, doch dauert das Durchsuchen der Listbox nach einem bestimmten Eintrag recht lange. Die folgende Routine liefert eine wesentlich schnellere Lösung über das Windows-API:

```
Declare Function SendMessage Lib "User" (ByVal hWnd As Integer, ByVal wMsg As
Integer, ByVal wParam As Integer, lParam As Any) As LongGlobal Const WM_USER =
&H400Global Const LB_SELECTSTRING = (WM_USER + 13)Sub SelectListItem (lst As
Control, Idx As String)  Dim i As Integer  i = SendMessage(lst.hWnd,
LB_SELECTSTRING, -1, ByVal Idx)End Sub
```

Angenommen, Sie haben ein Kombinationsfeld mit dem Namen Combo1, das unter anderem den Eintrag Test enthält, so lautet der Aufruf: SelectListItem Combo1, “Test”System über WIN.INI-Änderung informieren. Wenn eine Anwendung unter Windows Änderungen in der WIN.INI-Datei vornimmt, so weiss vor einem Neustart von Windows keine andere Anwendung von den Änderungen. Windows stellt nun ein API zur Verfügung, mittels dem alle Anwendungen dazu veranlasst werden können, die Einträge der WIN. INI neu einzulesen.

```
Const HWND_BROADCAST = &HFFFFConst WM_WININICHANGE = &H1ADeclare Function
SendMessage Lib "User" (ByVal hWnd As Integer, ByVal wMsg As Integer, ByVal wParam
As Integer, lParam As Any) As LongSub wininichangenotify ()  Dim i&  i& =
SendMessage(HWND_BROADCAST, WM_WININICHANGE, 0, 0)End Sub
```

Programmgesteuertes Booten

In Setuproutinen oder bei Werkzeugen zum Umstellen von Grafikauflösungen ist es oft nötig, Windows neu zu starten, damit vorgenommene Änderungen in Kraft treten. Manchmal ist es sogar unabdingbar, den Computer einem Warmstart zu unterziehen, um Treiberkonfigurationen oder ähnliches zu aktivieren. Die folgende Routine löst das Problem durch ein Windows-API:

```
Declare Function ExitWindows Lib "User" (ByVal RestartCode As Long,ByVal
DOSReturnCode As Integer) As IntegerSub ExitWin (ByVal nExitOption As Integer)  Dim
n As Integer  n = MsgBox("Hiermit beenden Sie Ihre Windows-Sitzung", 65, "Windows
beenden")  If n = 2 Then Exit Sub 'Benutzer wählte Nein  Select Case nExitOption
Case 1      n = ExitWindows(67, 0) 'Warmstart des Computers    Case 2      n =
ExitWindows(66, 0) 'Windows neu starten    Case 3      n = ExitWindows(0, 0)
'Windows verlassen  End SelectEnd Sub
```

Folgende Aufrufe führen zum entsprechenden Ziel:

```
Warmstart des Computers : ExitWin 1Windows neu starten : ExitWin 2Windows verlassen : ExitWin
3
```

Screenshot

Im folgenden sollen zwei ab und zu auftauchende Problemlösungen für das Entwicklerleben aufgezeigt werden. Das erste Problem ist das übernehmen eines beliebigen Fensterinhaltes oder des gesamten Bildschirmes in ein Bildfeld von Visual Basic. Das zweite ist das Ausdrucken des Inhaltes eines Bildfeldes. Mit diesen beiden Funktionen zusammen erstellen wir dann ein Werkzeug, mit dem Bildschirm-Schnappschüsse erstellt werden können.Um den Inhalt eines Fensters in ein Bildfeld zu übernehmen, benötigt man den Handle des entsprechenden Fensters (über das API GetDesktopWindow) erhält man den Handle des gesamten Bildschirms). Anhand dieses Handles bestimmt man den Quell-Devicekontext. Den Ziel-Devicekontext erhält man über die hDC-Eigenschaft des Bildfeldes. Mittels dem API BitBlt werden nun aus dem Quell-Devicekontext die Daten in den Ziel-Devicekontext übertragen. Damit alle Daten im Bildfeld Platz haben, wird es vorgängig noch auf die korrekte Grösse gebracht. Darauf bleibt nur noch den Quell-Devicekontext wieder freizugeben. Damit enthält das Bildfeld den gewünschten Fensterinhalt. Es ist zu beachten, dass die Eigenschaft AutoRedraw des Bildfeldes auf True gesetzt ist!

```
Type lrect  Left As Integer  Top As Integer  right As Integer  bottom As IntegerEnd
TypeDeclare Function GetDesktopWindow Lib "user" () As IntegerDeclare Function
```

```
GetDC Lib "user" (ByVal hWnd%) As IntegerDeclare Function BitBlt Lib "GDI" (ByVal
hDestDC%, ByVal X%, ByVal Y%, ByVal nWidth%, ByVal nHeight%, ByVal hScrDC%, ByVal
XSrc%, ByVal YScr%, ByVal dwRop&) As IntegerDeclare Function ReleaseDC Lib "User"
(ByVal hW_nd%, ByVal hDC%) As IntegerDeclare Sub GetWindowRect Lib "User" (ByVal
hWnd%, lpRect As lrect)Sub Screenshot (pic As PictureBox, hwndSrc%)  Dim hSrcDC%
Dim XSrc%, YSrc%  Dim nWidth%, nHeight%  Dim X%, Y%  Dim winSize As lrect  Dim
hDestDC%  Dim dwRop&  Dim suc%, dmy%    XSrc% = 0  YSrc% = 0  X% = 0  Y% = 0
pic.Top = 0  pic.Left = 0    hSrcDC% = GetDC(hwndSrc%)  GetWindowRect hwndSrc%,
winSize  nWidth% = winSize.right  nHeight% = winSize.bottom  hDestDC% = pic.hDC
pic.Width = nWidth% * screen.TwipsPerPixelX  pic.Height = nHeight% *
screen.TwipsPerPixelY  dwRop& = &HCC0020  suc% = BitBlt(hDestDC%, X%, Y%, nWidth%,
nHeight%, hSrcDC%, XSrc%, YSrc%, dwRop&)  dmy% = ReleaseDC(hwndSrc%, hSrcDC%)End
Sub
```

Nun können wir bereits mit dem Aufruf “Screenshot me.picture1, hWnd%” das durch den Handle hWnd% bestimmte Fenster in das Bildfeld picture1 laden.
Der Ausdruck eines Bildfeldes gestaltet sich so, dass zunächst die Eigenschaft ScaleMode sowohl des Bildfeldes als auch des Druckers auf “Pixel” gesetzt wird, weil das API StretchBlt Pixel-Koordinaten verlangt. Darauf wird ein Speicherbereich bereitgestellt, um das Bild für die Kopieraktion in den kompatiblen Devicekontext vorzubereiten. Über das API SelectObject wird das Objekt gespeichert. Mittels der Funktion StretchBlt wird nun das Bitmap vom Speicherbereich zum Drucker kopiert. Danach wird der Speicherbereich wieder freigegeben (Zuerst selektiert und dann gelöscht). Die folgende Routine erledigt diese Aufgabe:

```
Declare Function CreateCompatibleDC% Lib "GDI" (ByVal hDC%)Declare Function
SelectObject% Lib "GDI" (ByVal hDC%, ByVal hObject%)Declare Function StretchBlt%
Lib "GDI" (ByVal hDC%, ByVal X%, ByVal Y%, ByVal nWidth%, ByVal nHeight%, ByVal
hSrcDC%, ByVal XSrc%, ByVal YSrc%, ByVal nSrcWidth%, ByVal nSrcHeight%, ByVal
dwRop&)Declare Function DeleteDC% Lib "GDI" (ByVal hDC%)Declare Function Escape%
Lib "GDI" (ByVal hDC As Integer, ByVal nEscape As Integer, ByVal nCount As Integer,
LpInData As Any, LpOutData As Any)Sub printPicture (pic As Control)  Dim hMemoryDC%
Dim hOldBitMap%  Dim ApiError%  Dim prWidth, prHeight  Const SRCCOPY = &HCC0020
Const NEWFRAME = 1  Const PIXEL = 3  screen.MousePointer = 11  pic.Picture =
pic.Image  pic.ScaleMode = PIXEL  printer.ScaleMode = PIXEL  printer.Print " "  If
(printer.ScaleWidth - pic.ScaleWidth) < (printer.ScaleHeight - pic.ScaleHeight)
Then    prWidth = printer.ScaleWidth    prHeight = printer.ScaleHeight *
(pic.ScaleHeight / pic.ScaleWidth)  Else    prHeight = printer.ScaleHeight
prWidth = printer.ScaleWidth * (pic.ScaleHeight / pic.ScaleWidth)  End If
hMemoryDC% = CreateCompatibleDC (pic.hDC)  hOldBitMap% = SelectObject(hMemoryDC%,
pic.Picture)  ApiError% = StretchBlt(printer.hDC, 0, 0, prWidth, prHeight,
hMemoryDC%, 0, 0, pic.ScaleWidth, pic.ScaleHeight, SRCCOPY)  hOldBitMap% =
SelectObject(hMemoryDC%, hOldBitMap%)  ApiError% = DeleteDC(hMemoryDC%)  ' Falls
der Ausdruck nicht erfolgt, so entfernen Sie das Hochkomma  ' in der folgenden
Zeile!  'Debug.Print Escape(printer.hDC, NEWFRAME, 0, Null, Null)  printer.EndDoc
screen.MousePointer = 1End Sub
```

Nun ist es ein leichtes, unser Screenshot-Programm zu schreiben. Wir erzeugen eine Form die ein Bildfeld (Picture1) enthält, dessen AutoRedraw-Eigenschaft auf True und dessen Visible-Eigenschaft auf False gesetzt ist. Dann erstellen wir noch eine Schaltfläche (Command1) auf unserem Formular deren Caption-Eigenschaft wir auf “Screenshot” setzen. Die Form können wir so anpassen, dass sie genau dieselbe Grösse wie die Schaltfläche hat. In das Click-Ereignis der Schaltfläche schreiben wir nun noch folgenden Code:

```
Sub Command1_Click ()  Dim SrcHwnd%  Me.Visible = False  DoEvents  SrcHwnd% =
GetDesktopWindow()  Screenshot Me.Picture1, SrcHwnd%  printPicture Me.Picture1
Me.Visible = TrueEnd Sub
```

Kompilieren Sie das Programm und starten Sie es ausserhalb der Visual Basic-Entwicklungsumgebung. Die Form sorgt selbst dafür, dass sie nicht auf dem Screenshot angezeigt wird.Andreas Grob ist Ing. HTL Inf. und arbeitet als Programmierer/Analytiker bei der Firma Neuro Media AG

01 715 43 33 / FAX: 01 715 43 40

MS-DOS-Text in Windows-Text umwandeln
Der ANSI/ISO Zeichensatz von Windows weicht in den Zeichen über 127 stark vom PC-Zeichensatz ab. Dies erfahren deutschsprachige Anwender sehr schmerzhaft weil vorallem die schon aus früheren PC-Zeiten berüchtigten Umlaute wieder für Schwierigkeiten sorgen. Ein mit dem DOS-Editor Edit verfasster deutscher Text artet recht schnell in einen Hieroglyphen-Marathon aus. Zur Umwandlung bietet das Windows-API die Hand. Die Funktion OemToAnsi wandelt eine übergbebene Zeichenkette vom PC-Zeichensatz in den ANSI/ISO Zeichensatz von Windows um, während AnsiToOem die Rückumwandlung vornimmt. Beide Funktionen benötigen zwei Parameter wovon der erste die umzuwandelnde Zeichenkette und der zweite die Zielzeichenkette ist. Wie bei APIs mit Zeichenketten üblich muss die Zielzeichenkette mit der zurückerwarteten Anzahl Zeichen gefüllt sein, weil APIs nicht auf das Speichermanagement von Visual Basic zugreifen und einfach voraussetzen, dass der entsprechende Speicherplatz für die Zeichenkette zur Verfügung steht. Das folgende allgemeine Modul ermöglicht die Umwandlung.

```
Declare Function AnsiToOem Lib "Keyboard" (ByVal lpAnsiStr As String, ByVal
lpOemStr As String) As IntegerDeclare Function OemToAnsi Lib "Keyboard" (ByVal
lpOemStr As String, ByVal lpAnsiStr As String) As IntegerFunction
AnsiZuOem(Quelltext$) As String  Dim i%, Res$  Res$ = String$(Len(Quelltext$), " ")
i% = AnsiToOem(Quelltext$, Res$)  AnsiZuOem = Res$End FunctionFunction
OemZuAnsi(Quelltext$) As String  Dim i%, Res$  Res$ = String$(Len(Quelltext$), " ")
i% = OemToAnsi(Quelltext$, Res$)  OemZuAnsi = Res$End Function
```

Hoch-/Querformat-Umstellung per Programm
Die einfachste Möglichkeit, den Drucker von Hoch- auf Querformat umzustellen ist die, es dem Benutzer zu überlassen. Über den Standarddialog um den Drucker zu konfigurieren kann der Benutzer beliebige Druckereinstellungen selbst vornehmen. Um diese Möglichkeit zu nutzen muss man lediglich über das Menü File/Add File die Datei CMDIALOG.VBX einbinden, ein Common Dialog Steuerelement auf einem Formular erstellen und dessen Action-Eigenschaft bei Bedarf auf 5 (Printerdialog) setzen.
Leider hat dies nun aber auch Nachteile:
- Die Wiederherstellung der ursprünglichen Druckerkonfiguration ist nicht möglich.
- Die Einstellung wirkt sich auf alle Programme aus, die den Standarddrucker benutzen.
- Der Benutzer muss die Einstellung jedesmal manuell vornehmen.
Auf der Suche nach einem Weg, der oben genannte Nachteile nicht aufweist, stolperte ich im API -Dschungel über die Funktion ResetDC. Diese Funktion erlaubt es gezielt auf die Druckerkonfiguration Einfluss zu nehmen, ohne dass der Benutzer hinzugezogen werden müsste. Ausserdem bietet es folgende Vorteile:
- Die Umstellung kann von einer Seite zur anderen innerhalb desselben Dokumentes erfolgen.
- Die Umstellung gilt nur bis zum Ende des aktuellen Dokumentes (EndDoc).
- Die Umstellung wirkt sich nicht auf andere Programme aus.
Ich habe ein Modul mit zwei Routinen zusammengestellt, das die Umstellung des Druckers auf Hoch- rsp. auf Querformat ermöglicht. Um es einzugeben öffnen Sie zuerst ein neues Modul und definieren im Deklarationsteil folgenden Typ:

```
Type DEVMODE    ' 68 Bytes
   dmDeviceName As String * 32
   dmSpecVersion As Integer
   dmDriverVersion As Integer
   dmSize As Integer
   dmDriverExtra As Integer
   dmFields As Long
   dmOrientation As Integer
   dmpapersize As Integer
   dmPaperLength As Integer
   dmPaperWidth As Integer
   dmScale As Integer
   dmCopies As Integer
   dmdefaultsource As Integer
   dmPrintQuality As Integer
```

```
   dmColor As Integer
   dmDuplex As Integer
   dmYResolution As Integer
   dmTTOption As Integer
End Type
```

Diese Struktur nimmt die Konfiguration des Druckers auf. Dabei nimmt die Variable dmFields eine Sonderstellung ein, denn über diese Variable wird festgelegt, welche Einstellungen der Struktur berücksichtigt werden sollen. Dies ermöglicht es einzelne Einstellungen vorzunehmen, ohne die für die restlichen Parameter benötigten Werte zu kennen.
Nun wird noch das API selbst deklariert:

```
Declare Function ResetDC% Lib "GDI" (ByVal hDC%, lpdm As DEVMODE)
```

Darauf geben Sie die beiden Prozeduren Hochformat und Querformat ein:

```
Sub Querformat ()
 Dim dm As DEVMODE
 Dim i%
 dm.dmOrientation = 2 ' Setzt Querformat
 dm.dmFields = dm.dmFields Or 1 ' Flag DM_ORIENTATION
 i% = ResetDC%(printer.hDC, dm)
End Sub

Sub Hochformat ()
 Dim dm As DEVMODE
 Dim i%
 dm.dmOrientation = 1 ' Setzt Hochformat
 dm.dmFields = dm.dmFields Or 1 ' Flag DM_ORIENTATION
 i% = ResetDC%(printer.hDC, dm)
End Sub
```

Nun können Sie in Ihrem Programm durch Aufruf der entsprechenden Prozedur den Drucker temporär auf die gewünschte Papierorientation umstellen.

Drucker umschalten
Manchmal kommt es vor, dass der geplagte Programmierer sich vor dem Problem sieht, von seinem Programm aus auf einen anderen Drucker umzuschalten. Leider bietet Visual Basic nicht die Befehle Setz_Standarddrucker und Lies_Standarddrucker. Also müssen wir sie selbst schreiben. Zuerst noch ein Theorieblock der aber ganz kurz gehalten ist. Der Standarddrucker wird durch den Eintrag Device in der Sektion [windows] der Datei WIN.INI spezifiziert. Will man nun den Standarddrucker ändern, so muss man diesen Eintrag ändern und um die Änderung zu aktivieren, muss dem System noch bekanntgegeben werden, dass eine Änderung stattgefunden hat.Die Drucker, die auf dem System installiert worden sind, befinden sich in der Datei WIN.INI in der Sektion [devices].In dem Programm das wir nun erstellen, sollen alle möglichen Drucker in einer Liste aufgeführt werden und durch Auswahl des entsprechenden Druckers und Klick auf eine Schaltfläche soll der Standarddrucker festgelegt werden.Erstellen Sie nun ein Formular das wie folgt aussieht:

Nun deklarieren Sie die benötigten Windows API Funktionen im generellen Teil des Formulars:

```
' Windows API FunktionenDeclare Function GetProfileString Lib "Kernel" (ByVal
AppName$, ByVal KeyName As Any, ByVal Default$, ByVal ReturnedString$, ByVal
nSize%)Declare Function WriteProfileString Lib "Kernel" (ByVal AppName$, ByVal
KeyName$, ByVal lpString$)Declare Function PostMessageByString Lib "User" Alias
"PostMessage" (ByVal hWnd%, ByVal wMsg%, ByVal wParam%, ByVal lParam$)' Windows API
KonstantenConst HWND_BROADCAST = &HFFFFConst WM_WININICHANGE = &H1A
```

Das API GetProfileString liest aus der Datei WIN.INI den Wert des Topics in der gewünschten Sektion aus während WriteProfileString den Wert entsprechend setzt. Das API PostMessageByString wird dazu benutzt, um das System über die Änderung der Datei WIN.INI zu informieren, sodass die Werte der angegebenen Sektion neu eingelesen werden.Nun erstellen wir die Funktion Lies_Standarddrucker, die uns den Wert des Topics device aus der Sektion [windows] der Datei WIN.INI liefert:

```
Function Lies_Standarddrucker$ ()  Dim temp$, res%  'Herausfinden welcher Drucker
als Standarddrucker definiert ist  temp$ = String$(255, 0)  res% =
GetProfileString("windows", "device", "", temp$, Len(temp$))  Lies_Standarddrucker$
= temp$End Function
```

Die Prozedur Setz_Standarddrucker setzt den Topic device aus der Sektion [windows] der Datei WIN.INI auf den übergebenen Wert:

```
Sub Setz_Standarddrucker (Drucker$)  Dim res%  ' Neuen Defaultdrucker in die
Win.INI schreiben und das System  ' über die Änderung informieren  res% =
WriteProfileString("windows", "device", Drucker$)  res% =
PostMessageByString(HWND_BROADCAST, WM_WININICHANGE, 0, "windows")End Sub
```

Damit sind unsere neuen Befehle bereits fertig. Wir wollen nun die Liste der verfügbaren Drucker in unsere Listbox List1 einlesen. Dies soll die Prozedur Lies_Druckerliste erledigen:

```
Sub Lies_Druckerliste (Druckerliste As ListBox)  Dim res%, Drucker$, temp$,
Alle_Drucker$  Druckerliste.clear  Alle_Drucker$ = String$(4096, 0)  res =
GetProfileString("devices", 0&, "", Alle_Drucker$, Len(Alle_Drucker$))  If res <> 0
Then    Do      Drucker$ = Left$(Alle_Drucker$, InStr(Alle_Drucker$, Chr$(0)) - 1)
Alle_Drucker$ = Mid$(Alle_Drucker$, InStr(Alle_Drucker$, Chr$(0)) + 1)
' Einträge für den entsprechenden Drucker auslesen      temp$ = String$(255, 0)
res% = GetProfileString("devices", Drucker$, "", temp$, Len(temp$))
Druckerliste.AddItem Drucker$ + "," + temp$    Loop Until Left$(Alle_Drucker$, 1) =
Chr$(0)  End IfEnd Sub
```

Dadurch, dass in dem ersten Aufruf des API's GetProfileString als zweiter Parameter der Wert 0& übergeben wird, liefert das API alle vorhandenen Topics der angegebenen Sektion zurück. Die einzelnen Topics sind jeweils durch ein CHR$(0) voneinander getrennt. In der Schleife werden nun die einzelnen Werte der Topics in die übergebene Listbox abgefüllt.
Nun müssen wir dafür sorgen, dass beim Progammstart die Prozedur Lies_Druckerliste ausgeführt wird. Ausserdem soll der Standarddrucker angezeigt werden und die Liste auch auf den Standarddrucker positioniert werden:

```
Sub Form_Load ()  Dim i%  Lies_Druckerliste list1  label2.Caption =
Lies_Standarddrucker$()  For i% = 0 To list1.ListCount - 1    If list1.List(i%) =
label2.Caption Then      list1.ListIndex = i%      Exit For    End If  Next i%End
Sub
```

Zu guter Letzt schreiben wir im Click-Ereignis der Schaltfläche folgenden Code:

```
Sub Command1_Click ()    Setz_Standarddrucker (list1)  label2.Caption =
Lies_Standarddrucker$()End Sub
```

Damit können wir mit unserem Programm den Standarddrucker entsprechend der verfügbaren Drucker selbst umstellen und nach Herzenslust drucken.

Containerobjekte zwischen Formularen verschieben
Eine Rosine im API-Kuchen von Windows stellt die Funktion SetParent dar. Mittels dieser Funktion kann man Steuerelemente modularisieren. Visual Basic verfügt über sogenannte Containerobjekte. Ein Containerobjekt ist ein

Steuerelement innerhalb dem man weitere Steuerelemente plazieren kann. Containerobjekte erkennt man daran, dass bei dessen Veschiebung die darin enthaltenen Steuerelemente mit verschoben werden. So sind z.B. der Rahmen und das Picture Containerobjekte. In Visual Basic sind Containerobjekte selbständige Fenster mit eigenem Fensterhandle. Sie besitzen ausserdem ein übergeordnetes Fenster (Parent), nämlich das Formular (auch ein Fenster) innerhalb dem sie sich befinden rsp. angezeigt werden. Damit ist Endstation für Visual Basic. Nun kommt aber Windows zum Zug, denn es liefert ein API, das es ermöglicht festzulegen, welches Fenster das übergeordnete Fenster sein soll. Damit hat man eine Möglichkeit Containerobjekte zwischen Formularen zu verschieben und somit steht einer Modularisierung von Containerobjekten nichts mehr im Weg.Als Beispiel für ein allgemein nutzbares Containerelement wollen wir eine Uhr erstellen, die immer im aktiven Formular links oben angezeigt werden soll. Als Containerelement benutzen wir einen Rahmen.Erstellen Sie also in Visual Basic ein neues Projekt und plazieren Sie in der Form1 einen Rahmen (Frame1) in der linken oberen Ecke. In diesem Rahmen erzeugen Sie ein Bezeichnungsfeld (Label1). Damit haben wir das Containerobjekt mit seinem Inhalt erzeugt. Nun erstellen wir ein neues Modul und deklarieren im generellen Teil das API SetParent:

```
Declare Function SetParent% Lib "User" (ByVal hWndChild%, ByVal hWndNewParent%)
Im gleichen Modul schreiben wir nun die Prozedur ZeigUhr, die nichts anderes tut,
als das übergebene Formular zum Parent des Containerobjektes Frame1 zu machen..
Sub ZeigUhr (frm As Form)  Dim Res As Integer  Res = SetParent(Form1.Frame1.hWnd,
frm.hWnd)End Sub
```

Nun erzeugen Sie ein zweites Formular (Form2). Sobald dieses Formular aktiviert wird, so soll die Uhr auf diesem Formular angezeigt werden. Also schreiben Sie für die Form2 folgende Ereignisprozedur:

```
Sub Form_Activate ()  ZeigUhr MeEnd Sub
```

Hier bleibt noch zu bemerken, dass das Schliessen eines Formulars, das nicht der echte Parent eines enthaltenen Containerobjektes ist fatale Folgen hat. Deshalb muss das Formular Form2 noch mit folgender Ereignisprozedur ergänzt werden, die dafür sorgt, dass beim Entladen des Formulars das Containerobjekt wieder auf das richtige Formular gesetzt wird.

```
Sub Form_Activate ()  ZeigUhr MeEnd Sub
```

Damit nun bei der Rückkehr auf das erste Formular die Uhr wieder zurückkehrt, schreiben Sie für das Formular Form1 dieselbe Ereignisprozedur Form_Activate wie für das Formular Form2:

```
Sub Form_Activate ()  ZeigUhr MeEnd Sub
```

Jetzt müssen Sie noch dafür sorgen, dass das zweite Formular angezeigt wird, z.B. indem Sie folgende Ereignisprozedur für das Formular Form1 schreiben:

```
Sub Form_Load ()  Form2.ShowEnd Sub
```

Nun fehlt uns nur noch unsere Uhr. Dazu erzeugen Sie auf dem Formular Form1 einen Zeitmesser (Timer1) und stellen dessen Eigenschaft Interval auf 1000 (entspricht einer Sekunde). Für den Zeitmesser schreiben Sie folgende Ereignisprozedur die dem Bezeichnungsfeld in unserem Containerobjekt die aktuelle Uhrzeit zuweist.

```
Sub Timer1_Timer ()  label1.Caption = TimeEnd Sub
```

Beachten Sie hierbei, dass egal welches Fenster gerade das Parentfenster eines Containerobjektes ist, es immer nur über das Formular in dem es zur Entwicklungszeit erstellt wurde angesprochen wird.Nun können Sie das Programm starten und durch abwechselndes Anklicken der zwei Formen feststellen, wie die Uhr sich immer im aktiven Formular befindet. Jedes weitere Formular, welches ebenfalls die Uhr darstellen soll muss einfach dieselben Ereignisprozeduren enthalten wie unser Formular Form2.Damit steht einer allgemeinen Verwendung von Containerelementen nichts mehr im Weg. Es ist sogar denkbar eigene Containerelemente in fremden Fenstern zu plazieren. Dazu müsste lediglich der Handle des fremden Fensters bestimmt werden...

Shell modal
Ein sehr häufiges Problem für den Visual Basic Programmierer ist das Multitasking von Windows. Dies vor allem dann, wenn es eigentlich unerwünscht ist. Möchte man z.B. mit Visual Basic ein Programm schreiben, das mittels dem MS-DOS-Programm PKZIP.EXE Dateien komprimieren, dann die Grösse der komprimierten Datei bestimmen und falls die Dateilänge grösser ist als eine Diskette in einem Meldungsfenster diesen Umstand anzeigen soll, dann ist das leider mit Windows nicht so einfach zu realisieren. Man kann zwar PKZIP.EXE mit dem Befehl SHELL aufrufen, aber dann fährt das Programm weiter bevor PKZIP.EXE seine Arbeit beendet hat. Das Ziel ist es nun dafür zu sorgen, dass das aufrufende Programm auf den Abschluss des aufgerufenen Programmes wartet bis es weiterfährt wie etwa ein modales Dialogfenster.Um dies zu erreichen benötigen wir das Windows-API GetModuleUsage. Dieses API liefert einen Wert zurück, der aussagt wie oft das spezifizierte Modul geladen wurde. Die Deklaration des API's lautet:

```
Declare Function GetModuleUsage Lib "KERNEL" (ByVal InstanceID%) As Integer
```

Nun fehlt uns nur noch der ominöse Übergabeparameter InstanceID%. Dabei handelt es sich um den Handle der das entsprechende Modul spezifiziert. Unter einem Modul ist übrigens eine beliebige ausführbare Datei oder dynamische Linkbibliothek (DLL) zu verstehen. InstanceID% soll nun also unser aufgerufenes Programm spezifizieren. Nun ist es so, dass der Visual Basic Befehl SHELL mit dem wir das Programm starten als Rückgabewert gerade den Handle des gestarteten Programmes liefert. Dadurch benötigen wir kein weiteres API das uns den benötigten Handle liefert. Die folgende Prozedur dient als allgemeiner Ansatz das Problem zu lösen:

```
Sub ShellModal (Kommandozeile$)
 Dim Temp%,InstanceID%
 InstanceID% = Shell(Kommandozeile$, 3)
 Do
   Temp% = DoEvents()
 Loop Until GetModuleUsage(InstanceID%) = 0
End Sub
```

Das Programm erwartet als Übergabeparameter die Kommandozeile um das gewünschte Programm zu starten. Selbstverständlich funktioniert das Programm nicht nur bei DOS-Programmen sondern auch bei Windows-Programmen. In der Variablen InstanceID% wird der Modul-Handle des gestarteten Programmes gespeichert. In einer DoEvents-Schleife wird darauf gewartet, dass der Rückgabewert von GetModuleUsage 0 beträgt. Erst dann wird die Prozedur beendet.

Mauszeiger verstecken
Windows bietet ein API um den Mauszeiger anzuzeigen oder zu verstecken. Manche Abstürze von Programmen lassen gleich auch den Mauszeiger verschwinden. Es bleibt nichts anderes übrig, als Windows zu beenden und wieder zu starten um den Mauszeiger wieder anzuzeigen. Nicht so, wenn man ein Programm schreibt, welches den Mauszeiger wieder einschaltet. Genauso kann es unter Umständen erwünscht sein, den Mauszeiger zu verstecken wie z.B. bei einem Bildschirmschoner. Mit der folgenden Routine lassen sich beide Varianten realisieren.

```
Declare Function ShowCursor Lib "User" (ByVal bShow As Integer) As Integer
Const Ein = True
Const Aus = False

Sub Mauszeiger (Modus%)
 Dim i%
 i% = ShowCursor(Modus%)
End Sub
```

Durch den Aufruf Mauscursor Ein kann nun der Mauszeiger eingeschaltet werden, mit Mauscursor Aus kann er ausgeschaltet werden.

Schriften vertikal ausgeben

Auch wenn man sich im allgemeinen nicht mit den komplizierten Ausgabemechanismen für Schriftarten unter Windows auseinandersetzten möchte, kommt man manchmal gar nicht darum herum. Ich wollte ein Zeichenprogramm erstellen mit einem vertikalen Lineal links. In den Programmen, die ich als Vorbild genommen hatte, war das Lineal vertikal beschriftet. Visual Basic ermöglicht standardmässig nur die horizontale Zeichenausgabe. Um trotzdem in der Lage zu sein, meinem Lineal einen professionellen Touch zu geben, stürzte ich mich in die Tiefen der Windows-APIs um Schriftarten vertikal auszugeben.
Dabei hatte ich zwei wichtige Erkenntnisse:
- Rasterschriften können nicht rotiert werden (Alle TrueType-Schriften können rotiert werden)
- Die Funktionalität der Eigenschaft AutoRedraw von Bildfeldern muss speziell behandelt werden, da die Windows-APIs ihre Informationen nicht automatisch in das von Visual Basic gespeicherte Bitmap übertragen.
Ansonsten befolgte ich die Schritte eine Schriftart auszugeben, wie sie in der einschlägigen Fachliteratur beschrieben sind. So zaubert man nach Schema F jede TrueType-Schrift in der gewünschten Art und Weise auf den Bildschirm oder falls gewünscht auf den Drucker, wenn statt dem Bildfeld das Druckerobjekt Printer angegeben wird.
Um die Schrift zu rotieren, benötigt man nur die Strukturvariable lfEscapement aus der Struktur LOGFONT. Diese nimmt den gewünschten Winkel auf, um den die Schrift rotiert werden soll und zwar in 1/10° von 0 bis 3600. Um den Text vertikal auszugeben, muss lfEscapement auf 900 gesetzt werden. Wichtig ist noch, dass die Ausgabekoordinaten sich auf die linke obere Ecke des nicht rotierten Textes bezieht. Ist der Text jedoch um 90 Grad rotiert, so ist dies die Ecke links unten!
Damit kann der Text bereits vertikal ausgegeben werden. Ein Bildfeld, das ein Lineal enthält, hat nun aber mit Vorteil die Eigenschaft AutoRedraw eingeschaltet, um ein möglichst schnelles Bildschirmrollen zu ermöglichen. Ist die Eigenschaft AutoRedraw true, dann werden alle über Visual Basic Methoden ausgegebenen Informationen in einen Zwischenspeicher transferiert, und erst dann wird dieser in das Bildfeld übertragen. Da das API TextOut an diesem Mechanismus vorbeiläuft muss man sich eines Tricks bedienen, um die angezeigten Informationen in den Zwischenspeicher zu speichern. Bei einem Bildfeld bezeichnet die Eigenschaft Image das sichtbare Bild und die Eigenschaft Picture das von Visual Basic zwischengespeicherte Abbild. Indem nun die Eigenschaft Image der Eigenschaft Picture zugewiesen wird, wird das Abbild entsprechend dem sichtbaren Bild aktualisiert.
Mit dem folgenden allgemeinen Modul kann Text vertikal ausgegeben werden. Der Aufruf lautet wie folgt:
WriteVertical (PicText as PictureBox, X!, Y!, Text$)
Parameter:
PicText: Bildfeld, in dem der Text ausgegeben werden soll
X! : X-Koordinate der linken unteren Ecke des auszugebenden Textes
Y! : Y-Koordinate der linken unteren Ecke des auszugebenden Textes
Text$: Text der vertikal ausgegeben werden soll
Die Prozedur berücksichtigt automatisch die Einstellung der AutoRedraw-Eigenschaft des übergebenen Bildfeldes.

Schritt für Schritt Anleitung:

Erstellen Sie ein neues Projekt und erzeugen Sie in der Form das Bildfeld Picture1, das den gewünschten Text anzeigen soll. Setzen Sie die Eigenschaft ScaleMode des Bildfeldes auf Pixels (3) und wählen Sie als FontName eine TrueType-Schriftart. Daneben erzeugen Sie eine Schaltfläche Command1. Wenn diese Schaltfläche angeklickt wird, soll der Text Hallo in dem Bildfeld ausgegeben werden.
Dazu schreiben Sie folgenden Code in die entsprechende Ereignisroutine der Form1:

```
Sub Command1_Click ()
 WriteVertical picture1, 1, picture1.ScaleHeight - 1, "Hallo"
End Sub
```

Dann erstellen Sie ein Modul und fügen folgenden Code in das Modul ein (Tip: kopieren Sie die Konstanten, Typen und Deklarationen aus der Hilfedatei WIN31API.HLP im Unterverzeichnis WINAPI von Visual Basic):

```
Global Const LF_FACESIZE = 32

Type TEXTMETRIC
```

```
   tmHeight As Integer
   tmAscent As Integer
   tmDescent As Integer
   tmInternalLeading As Integer
   tmExternalLeading As Integer
   tmAveCharWidth As Integer
   tmMaxCharWidth As Integer
   tmWeight As Integer
   tmItalic As String * 1
   tmUnderlined As String * 1
   tmStruckOut As String * 1
   tmFirstChar As String * 1
   tmLastChar As String * 1
   tmDefaultChar As String * 1
   tmBreakChar As String * 1
   tmPitchandFamily As String * 1
   tmCharSet As String * 1
   tmOverhang As Integer
   tmDigitizedAspectX As Integer
   tmDigitizedAspectY As Integer
End Type

Type logfont
   lfHeight As Integer
   lfWidth As Integer
   lfEscapement As Integer
   lfOrientation As Integer
   lfWeight As Integer
   lfItalic As String * 1
   lfUnderline As String * 1
   lfStrikeOut As String * 1
   lfCharSet As String * 1
   lfOutPrecision As String * 1
   lfClipPrecision As String * 1
   lfQuality As String * 1
   lfPitchandFamily As String * 1
   lfFaceName As String * LF_FACESIZE
End Type

Type rect
   left As Integer
   top As Integer
   right As Integer
   bottom As Integer
End Type

Declare Function CreateFontIndirect Lib "GDI" (lpLogFont As logfont) As Integer
Declare Function DeleteObject Lib "GDI" (ByVal hObject As Integer) As Integer
Declare Sub GetClientRect Lib "User" (ByVal hWnd As Integer, lpRect As rect)
Declare Function SelectObject Lib "GDI" (ByVal hDC As Integer, ByVal hObject As
Integer) As Integer
Declare Function TextOut Lib "GDI" (ByVal hDC As Integer, ByVal X As Integer, ByVal
Y As Integer, ByVal lpString As String, ByVal nCount As Integer) As Integer
Declare Function GetTextMetrics Lib "GDI" (ByVal hDC As Integer, lpMetrics As
```

```
TEXTMETRIC) As Integer

Sub WriteVertical (PicText As PictureBox, X!, Y!, Text$)
   Dim tm As TEXTMETRIC
   Dim r$
   Dim crlf$
   Dim oldfont%
   Dim tbuf As String * 80
   Dim FontToUse%
   Dim lf As LOGFONT
   Dim oldhdc%
   Dim rc As RECT
   Dim di%

   di% = GetTextMetrics(PicText.hDC, tm)
   crlf$ = Chr$(13) + Chr$(10)
   If FontToUse% <> 0 Then di% = DeleteObject(FontToUse%)
   lf.lfHeight = tm.tmHeight * 1.1
   lf.lfWidth = tm.tmAveCharWidth * .9
   lf.lfEscapement = 900
   lf.lfWeight = tm.tmWeight
   lf.lfItalic = tm.tmItalic
   lf.lfUnderline = tm.tmUnderlined
   lf.lfStrikeOut = tm.tmStruckOut
   lf.lfOutPrecision = Chr$(0)
   lf.lfClipPrecision = Chr$(0)
   lf.lfQuality = Chr$(0)
   lf.lfPitchAndFamily = tm.tmPitchAndFamily
   lf.lfCharSet = tm.tmCharSet
   lf.lfFaceName = PicText.FontName + Chr$(0)
   FontToUse% = CreateFontIndirect(lf)
   If FontToUse% = 0 Then Exit Sub
   oldhdc% = SelectObject(PicText.hDC, FontToUse%)
   GetClientRect PicText.hWnd, rc
   di% = TextOut(PicText.hDC, X!, Y!, (Text$ + Chr$(0)), Len(Text$))
   di% = SelectObject(PicText.hDC, oldhdc%)
   If FontToUse% <> 0 Then di% = DeleteObject(FontToUse%)
   If PicText.AutoRedraw Then PicText.Picture = PicText.Image

End Sub
```

Wenn Sie nun das Programm starten und auf die Befehlsschaltfläche klicken wird im Bildfeld vertikal Hallo ausgegeben, vorausgesetzt die Eigenschaft FontName des Bildfeldes enthält den Namen einer Vektorschrift (Rasterschriften können nicht rotiert werden).

Hotkey

Es ist mit einem Visual Basic Programm sehr komfortabel möglich die Tastatur zu überwachen, solange das Programm den Focus besitzt. Aber wehe es sollen auch Tastendrücke abgefangen werden, wenn das Programm den Focus nicht hat. Angenommen es stellt sich das Problem, dass eine Visual Basic Adressverwaltung resident geladen ist und mittels der Tastenkombination Ctrl+A aufgerufen werden soll, wenn man z.B. in Write einen Brief erfasst, so hat man Mühe dies mit Visual Basic Bordmitteln zu realisieren.
Hierbei hilft das Windows-API GetAsyncKeyState. Mit diesem API kann bestimmt werden, ob seit dem letzten Aufruf von GetAsyncKeyState eine bestimmte Taste gedrückt worden ist, und ob diese Taste im Augenblick gerade noch gedrückt ist.
Die Definition des API lautet:Declare Function GetAsyncKeyState Lib “User” (ByVal vKey%)

Das API benötigt als Parameter den virtuellen Tastenkode dessen Status überprüft werden soll. Die entsprechenden Konstanten beginnen mit VK_... und Sie finden die Konstanten in der Hilfedatei win31api.hlp unter “Global Constants”. Der Rückgabewert des API hat das Bit 0 gesetzt, wenn die Taste seit dem letzten Aufruf von GetAsyncKeyState gedrückt wurde und das Bit 15 ist gesetzt, wenn die Taste im Augenblick der Abfrage gedrückt war.
Im folgenden Beispiel soll ein Formular per Hotkey falls es minimiert ist wieder geöffnet werden und falls es unsichtbar ist wieder angezeigt werden. Mittels zwei Kombinationslisten kann der gewünschte Hotkey ausgewählt werden, wobei die Listen nur eine Auswahl der möglichen Tastenkodes enthalten. Über ein Timer-Steuerelement wird alle 500 Millisekunden das API GetAsyncKeyState aufgerufen und falls die Tasten gedrückt wurden das Formular angezeigt:

Erstellen Sie ein Formular das zwei Kombinationslisten Combo1 und Combo2 ,eine Schaltfläche Command1 und einen Timer Timer1 enthält, dessen Intervall z.B. auf 500 eingestellt ist (siehe Bild 1).
Geben Sie nun das Listing 1 ein.

```
Declare Function GetAsyncKeyState Lib "User" (ByVal vKey As Integer) As Integer

Sub Add2Combo (cmb As combobox, inhalt$, wert%)
 cmb.AddItem inhalt$
 cmb.itemdata(cmb.newindex) = wert%
End Sub

Sub Form_Load ()
 Dim i%
 Add2Combo Combo1, "(keine)", &H0
 Add2Combo Combo1, "Shift", &H10
 Add2Combo Combo1, "Ctrl", &H11
 Add2Combo Combo1, "Alt", &H12
 Combo1.ListIndex = 0

 For i% = Asc("A") To Asc("Z")
   Add2Combo Combo2, Chr$(i%), i%
 Next i%
 For i% = 1 To 24
   Add2Combo Combo2, "F" & Format$(i%), 111 + i%
 Next i%
 Combo2.ListIndex = 0
End Sub

Sub Timer1_Timer ()
 Dim ctrl%, taste%
 If Me.Combo1.ItemData(Me.Combo1.ListIndex) = 0 Then
   ctrl% = 1
 Else
   ctrl% = GetAsyncKeyState(Me.Combo1.ItemData(Me.Combo1.ListIndex))
 End If
 taste% = GetAsyncKeyState(Me.Combo2.ItemData(Me.Combo2.ListIndex))
 If ctrl% <> 0 And taste% <> 0 Then
   Me.Show
   If Me.WindowState = 1 Then
     Me.WindowState = 0
   End If
 End If
End Sub

Sub Command1_Click ()
```

```
 Me.Hide
End Sub
```

Debug-Fenster löschen

Wie komfortabel ist es doch über den Befehl Debug.Print Debugginginformationen im Debugfenster auszugeben. Doch nach ein zwei Läufen steht man bald einmal vor dem Problem, welche Ausgaben zum vorhergehenden Lauf und welche zum aktuellen gehören. Natürlich kann man jedesmal vor Programmstart den Inhalt des Debugfensters markieren und mit der Deletetaste löschen. Doch wer denkt schon jedesmal daran. Also soll die Arbeit doch vom Programm übernommen werden! Die folgende kleine Routine übernimmt die Aufgabe. Wird sie z.B. im Form_Load-Ereignis aufgerufen, so löscht sie den gesamten Inhalt des Debugfensters.

```
Declare Function FindWindow Lib "User" (ByVal lpClassName As Any, ByVal
lpWindowName As Any) As Integer
Declare Function SendMessage Lib "User" (ByVal hWnd As Integer, ByVal wMsg As
Integer, ByVal wParam As Integer, lParam As Any) As Long
Declare Function GetWindow Lib "User" (ByVal hWnd%, ByVal wCmd%) As Integer

Sub Debug_Clear ()
 Dim hWndVB%, hWndDebug%, res%
 ' bestimmen des Handles des Debug-Fensters (Fensterklasse = OFEDT)
 hWndVB% = FindWindow("OFEDT", 0&)
 ' Wurde das Fenster nicht gefunden läuft das Programm nicht in der
 ' Entwicklungsumgebung
 If hWndVB% = 0 Then Exit Sub
 ' bestimmen des Handles des Kindfensters (Das Textfenster des Debug-Fensters)
 hWndDebug% = GetWindow(hWndVB%, 5)
 ' Wurde kein Fensterhandle gefunden, existiert das Debug-Fenster nicht
 If hWndDebug% = 0 Then Exit Sub
 ' Text des Debug-Fensters auf eine Nullzeichenkette setzen
 res% = SendMessage(hWndDebug%, &HC, 0, 0&)
End Sub
```

Inhalt Beispieldiskette

Verzeichnis	Beschreibung	API Funktion
1. Anltng	Beschreibung der API Funktionen	
2. Bitblt	Grafische Anzeige von Bitmaps	BitBlt StretchBlt
3. Clipreg	Bestimmte Bereiche in einer Form mit Grafik füllen	SelectClipRgn GetClientRect CreateEllipticRgnIndirect
4. Cursor	Cursorform auf Knopfdruck ändern	GlobalLock GlobalUnLock CreateCursor GetWindowWord SetClassWord GetClassWord GetBitmapBits
5. Devcont	Form oder Printer Context anzeigen	GetDeviceCaps
6. Dragdrop	Beispiel von Drag and Drop	DragAcceptFiles PeekMessage DragQueryFile DragFinish
5. Drucker2	Aktuellen Drucker wechseln	GetProfileString GetPrivateProfileString WriteProfileString WritePrivateProfileString
6. Exeicon	Programme über VB Form ausführen	GetModuleUsage
7.Fltask	Alle Objekte auf dem Bildschirm in Bewegung bringen	GetWindow GetWindowText GetWindowTextLength GetDeviceCaps FindWindow MoveTo MoveWindow
8. Font	Informationen über Schriftarten anzeigen	SetMapMode GetMapMode
9. Log	Texteditor um kleinere Textdateien zu editieren	GetModuleFileName GetClassWord GetActiveWindow isWindow GetWindowTask

10. Lstsuch	Schnelles suchen nach einem Eintrag in einer List Box	SendMessage GetCurrentTime
11. Maskdemo	Zwei Bitmaps übereinander in neuen Bereich kopieren	BitBlt
12. Menu	Darstellen von Bitmaps in einem Menu	GetMenu GetsubMenu GetMenuItemID ModifyMenu GetMenuItemBitmaps
13. Nomdi	Beispiel einer Toolbox die immer im Vordergrund bleibt	SetParent
14. Puzzle	Aus einem bestehenden Bitmap wird ein Puzzle erstellt	GetObjectAPI CreateDIBitmap CreatePatternBrush GetClientRect CreateCompatibleDC StretchBlt PatBlt SelectObject
15 Q2\Drucker	Feststellen was für Drucker zur Verfügung stehen und den Default wechseln	GetProfileString WriteProfileString PostMessageByString
16 Q2\handle	Feststellen welche Windows Applikationen aktiv sind und Informationen über diese Applikationen anzeigen	GetParent GetWindow GetWindowWord FlashWindow GetModuleFileName GetClassName GetDeskTopWindow ShowWindow ApiSetFocus
17 Q2\Point	Zeigt die Adresse einer Variablen im Speicher an	PointerToObject DeRefString
18 Q3\Sysinfo	Liefert System Informationen	RemoveMenu GetVersion GetNumTasks GetFreeSpace GetCurrentTime

		GetSubMenu GetSystemMenu GetWinFlags
19. Setparent	Bewegt ein Frame von einer Form in eine andere	SetParent
20. Sysinfo	Zeigt System Informationen an	GetWinFlags GetFreeSystemResources GetFreeSpace
21. Task	Anzeigen der aktiven Applikationen	GetWindow GetWindowText GetWindowTextLength
22. Textvert	Schrift vertikal darstellen	DeleteObject CreateFontIndirect GetClientRect SelectObject TextOut GetTextMetrics
23. Tools\Apix	Zusammenfassung aller API Funktionen	GetFocus SendMessage
24. Top	Form die immer im Vordergrund bleibt	GetVersion SetWindowPos
25. Waitdos	Dos Befehle ausführen	GetModuleUsage