VFD-Studio

Dokumentation der Klasse TPortIOVFD
und der Komponente vfd

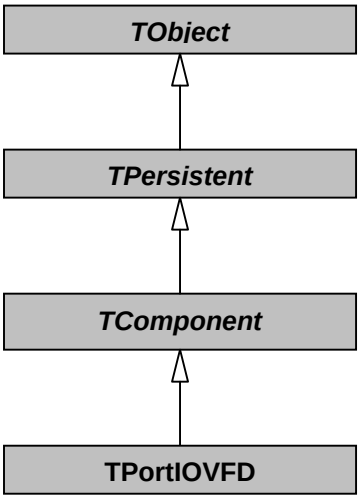
Inhalt

Hierarchie / Vererbung

Die DLPortIO-Komponente und der LPT-Treiber bilden die letzte Ausgabestufe der Kommunikation mit dem Display.
Den wichtigsten Teil der Kommunikation mit dem Display übernimmt jedoch die Komponente *vfd*, ein Objekt der Klasse TPortIOVFD.
Der Sourcecode dieser Klasse befindet sich in dem Listing PortIOVFD.pas

Die Klasse TPortIOVFD ist abgeleitet von TComponent, welche eine abstrakte Klasse ist und das grundlegende Verhalten, das allen Komponenten in Delphi gemeinsam ist kapselt.
TComponent wiederum ist abgeleitet von den Basisklassen TPersistent und TObject.

Klassenhierarchie:



TComponent wurde als Oberklasse für diese Komponente gewählt, da es sich bei TPortIOVFD um eine nicht-visuelle Komponente handelt, die in der Komponentenpalette angezeigt und im Formular-Designer bearbeitet werden soll.

Folgend nun die nähere Beschreibung dieser Klasse.

Übersicht - TPortIOVFD:

vfd: TPortIOVFD
- BA: word - DC: array[0..9, 0..191] of byte - LightLevel: byte - FOut: boolean - framewidth: byte - frame, frames, Xposit : word - Bmap: TFileName - Digital: boolean - Xuhr: byte - Bmp, ClockBitmap: TBitmap - FadeTimer, AnimateTimer, ClockTimer: TTimer + PortIOComponent: TDLPortIO + LPTPort: word + Bitmap: TBitmap
outport(Adresse: word, wert, byte) # Doppelpunkt() # LoadChars() # AnimateTimerEvent(Sender: TObject) # AnimateBitmap(bmp: TBitmap; xPos, Frame, FrameWidth: word; Farbe: TColor); # FadeTimerEvent(Sender: TObject) # ClockTimerEvent(Sender: TObject) # RefreshTimer() # BitmapChangeEvent(Sender: TObject) + constructor Create(AOwner: TComponent) + destructor Destroy() + InitVFD() + WriteCommand(c: byte) + WriteData(d: byte) + SetCursor(C_low, C_high: byte) + ClearScreen(ScreenNr: byte) + SelectScreen(Screen: byte) + PaintString(s: string; col, row: byte) + PaintBitmapRect(Source: TBitmap; xDisplay,yDisplay,width,xOffset: byte; Farbe: TColor) + ClearBitmap() + SetPixelbyte(Pixelbyte,x,y: byte) + PauseAnimation() + StopAnimation() + Fade(off: boolean; Speed: word) + Time(analog: boolean; Xposition: byte) + stopClock()

Kommunikation mit LPT-Treiber

Die Kommunikation mit dem LPT-Treiber wurde wie bereits erwähnt über die DLPortIO-Komponente abgewickelt.
Damit die vfd-Komponente mit dieser kommunizieren kann, benötigt sie eine Referenz auf diese Komponente.
Dies wurde einfach durch eine public-Deklaration `PortIOComponent: TDLPortIO;` realisiert.
Beim Starten des Hauptprogrammes weist dieses dann der `PortIOComponet`-Variable der vfd-Komponente die PortIO-Komponente zu, so dass `vfd` auf `DLPortIO` zugreifen kann:

```
vfd.PortIOComponent:=Dlportio;  //PortIO-Komponente für VFD-Komponente zuweisen
dlportio.DriverPath:=extractfilepath(application.exename);
vfd.LPT:=mainform.LPTPort;      //LPT zuweisen
```

Die PortIOVFD-Klasse verfügt zur Kommunikation mit dem Treiber über die PortIO-Komponente nun noch folgende Prozedur:

```
// Outport
// Sendet Daten an LPT-Adresse über PortIO-Komponente
// Parameter:
//           Adresse: word - LPT-Adresse (BA für D0-D7, BA+2 für Steuerleitungen)
//           wert: byte    - auszugebenden Daten
procedure TPortIOVFD.outport(Adresse: word; wert: byte);    { Portausgabe }
var
   DataWrite : Longword;  //Daten
begin
   DataWrite:=wert;
   try
      DataWrite:=Longword(DataWrite);  //Umwandlung in Longword (falls nötig)
   except
   end;
   try
    PortIOComponent.Port[Adresse]:=Byte(DataWrite and $FF);
   except
   end;
end;
```

Ein Schreibzugriff auf das Property „Port“ der PortIOComponent bewirkt eine umgehende Ausgabe an der spezifizierten Adresse.

Kommandos

Das Display kennt verschiedene „Commands“, welche durch unterschiedliche Bitkombinationen der Datenleitungen (D_0-D_7) definiert sind.
Die Beschreibung der Commands steht im Displaydatenblatt auf den Seiten 4-7.

Hier die Liste der möglichen Commands:

Command					Daten								Funktion
D_4	D_3	D_2	D_1	D_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
0	0	0	0	0	-								Beide Screens aus
0	0	0	0	1	-								Screen1 an
0	0	0	1	0	-								Screen 2 an
0	0	0	1	1	-								Beide Screens an
0	0	1	0	0	-								Cursor läuft automatisch weiter
0	0	1	0	1	-								Cursor hält automatisch
0	0	1	1	0	-								Screen2 ist Charakterscreen
0	0	1	1	1	-								Screen2 ist Grafikscreen
0	1	0	0	0	x	x	x	x	x	x	x	X	Daten schreiben
0	1	0	0	1	x	x	x	x	x	x	x	X	Daten lesen
0	1	0	1	0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Untere Startadresse Screen1
0	1	0	1	1	x	x	x	A_{12}	A_{11}	A_{10}	A_9	A_8	Obere Startadresse Screen1
0	1	1	0	0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Untere Startadresse Screen2
0	1	1	0	1	x	x	x	A_{12}	A_{11}	A_{10}	A_9	A_8	Obere Startadresse Screen2
0	1	1	1	0	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Untere Startadresse des Cursor
0	1	1	1	1	x	x	x	A_{12}	A_{11}	A_{10}	A_9	A_8	Obere Startadresse des Cursor
1	0	0	0	0	-								Screen 1&1 OR-verknüpfen
1	0	0	0	1	-								Screen 1&1 EXOR-verknüpfen
1	0	0	1	0	-								Screen 1&1 AND-verknüpfen
1	1	0	0	0	-								100% Helligkeit
1	1	0	0	1	-								87.5% Helligkeit
1	1	0	1	0	-								75% Helligkeit
1	1	0	1	1	-								62,5% Helligkeit

Um ein Command an das Display zu senden muss zunächst das *Command/Data Select Signal (C/D)* auf logisch high gesetzt werden. D.h. die *Selec*t-Leitung muß logisch high sein.
Um nun beispielsweise die Helligkeit auf 75% einzustellen genügt es auf dem LPT-Datenregister den Wert 26 (11010_2) auszugeben, das *WriteSignal (/WR)* an der *Strobe*-Leitungung kurz auf high zu setzen und dann wieder */WR* und *C/D* auf low.
Die Reihenfolge also:

C/D auf high, D_4-D_0 auf 11010_2, */WR* auf high, */WR* auf low, *C/D* auf low

Um z.B. Daten zu schreiben:

erst Command „Daten schreiben“ nach obiger Reihenfolge senden,
dann *C/D* auf low, D_7-D_0 auf gewünschten Wert, */WR* auf high, */WR* wieder auf low, *C/D* wieder auf high

Hinweis:
/WR auf high bedeutet, dass die *Strobe*-Leitung logischen Highpegel hat - */WR* ist, wie das „/“ andeutet, jedoch ein invertierender Signaleingang, weshalb der high-low Wechsel der *Strobe*-Leitung seitens des Displaycontrollers eine 0->1 Signalflanke bewirkt und dadurch eine Übernahme der an D_7-D_0 anliegenden Daten (vlg. auch Displaydatenblatt S. 9 Abschn. 13)

Realisierung mit Delphi (PortIOVFD.pas)

Anhand der Definitionen der Commands und der Reihenfolge, in der die Steuerleitungen auf high/low zu setzen sind, konnten daraus zwei Prozeduren zur Kommunikation erstellt werden, auf denen alle Displayausgaben basieren:

```
procedure TPortIOVFD.WriteCommand(c: byte);
begin
     // setze C/D auf 1 für "C" - Command
     outport(BA+2,(CnD+nRD) XOR Maske);
     outport(BA+2,(CnD+nRD) XOR Maske);
     // lege Kommandobyte an Datenleitungen
     outport(Ba,c);
     outport(Ba,c);
     // setze /WR auf 1
     outport(BA+2,(CnD+nWR+nRD) XOR Maske);
     outport(BA+2,(CnD+nWR+nRD) XOR Maske);
     // /WR wieder rücksetzen
     outport(BA+2,(CnD+nRD) XOR Maske);
     outport(BA+2,(CnD+nRD) XOR Maske);
     // setze C/D wieder auf "D"- Data
     outport(BA+2,(nRD+nWR) XOR Maske);
     outport(BA+2,(nRD+nWR) XOR Maske);
end;
```

```
procedure TPortIOVFD.WriteData(d: byte);
begin
     // setze C/D auf "D" - Data
     outport(BA+2,(nRD+nWR) XOR Maske);
     outport(BA+2,(nRD+nWR) XOR Maske);
     // lege Datenbyte an Datenleitungen an
     outport(BA,d);
     outport(BA,d);
     // /WR wieder zurücksetzen
     outport(BA+2,(nRD) XOR Maske);
     outport(BA+2,(nRD) XOR Maske);
     // setze C/D wieder auf "C" - Command
     outport(BA+2,(CnD+nRD+nWR) XOR Maske);
     outport(BA+2,(CnD+nRD+nWR) XOR Maske);
end;
```

Erläuterungen:

BA bezeichnet die Basisadresse des LPTs. BA ist eine private Variable der PortIOVFD-Klasse und wird durch das Property LPT geschrieben/gelesen.
outport ist die unter *Kommunikation mit LPT-Treiber* beschriebene Prozedur zur Datenausgabe über die PortIO-Komonente.
Die Steuerleitungen, bzw. ihre Wertigkeit wurden innerhalb der Klasse als Konstanten definiert:

```
const
{ Konstanten für Zugriff auf Steuerleitungen }
     nWR = 1;       { Wertigkeit /WR @ STROBE   }
     nRD = 2;       { Wertigkeit /RD @ AUTOFEED }
     CnD = 8;       { Wertigkeit C/D @ SELECT   }
     Maske = 11;    { XOR-Bitmaske für Ausgabe auf Steuerleitungen }
```

Die Steuerleitungen (Strobe, Autofeed, Select, Init) sind zum Teil invertierend.
Mit den obigen Definitionen ist jedoch ein einfacher und unkomplizierter Umgang mit den high/low-Pegeln möglich.
Wenn z.B. */WR* und *C/D* auf high und */RD* auf low gesetzt werden sollen, so genügt es die Wertigkeiten der high-Leitungen zu addieren (1_2+1000_2) und mit der XOR-Maske Maske (1011_2) zu verknüpfen. Der resultierende Wert (0010_2) entspricht dann dem Wert, der ausgegeben werden muss, um das gewünschte Ergebnis zu erhalten.

Definitionen globaler Konstanten
Die Commandos wurden im Sourcecode ebenfalls als globale Konstanten definiert:

```
{ Kommandodefinitionen, Werte in Hexadezimaldarstellung}
     CMD_SCR_OFF =     $00;    { Beide Screens aus }
     CMD_SCR1_ON =     $01;    { Screen 1 an }
     CMD_SCR2_ON =     $02;    { Screen 2 an }
     CMD_SCR1and2_ON = $03;    { Beide Screens an }

     CMD_CUR_INC =     $04;    { Cursor fährt automatisch weiter *DEFAULT }
     CMD_CUR_HOLD =    $05;    { Cursor stoppt }

     CMD_SCR2_CHAR =   $06;    { Screen 2 als Textdisplay }
     CMD_SCR2_GFX =    $07;    { Screen 2 als Grafikdisplay }

     CMD_WRITE_DATA =  $08;    { Schreibkommando }
     CMD_READ_DATA =   $09;    { Lesekommando }

     CMD_SCR1_lowADR =  $0A;   { Kommando: untere Adresse Screen 1 }
     CMD_SCR1_highADR = $0B;   { Kommando: obere Adresse Screen 1  }
     CMD_SCR2_lowADR =  $0C;   { Kommando: untere Adresse Screen 2 }
     CMD_SCR2_highADR = $0D;   { Kommando: obere Adresse Screen 2  }

     CMD_CUR_lowADR =   $0E;   { Kommando: untere Adresse Cursor }
     CMD_CUR_highADR =  $0F;   { Kommando: obere Adresse Cursor  }

     CMD_SCR_OR =       $10;   { ODER-Verknüpfung }
     CMD_SCR_XOR =      $11;   { NICHT-ODER-Verknüpfung }
     CMD_SCR_AND =      $12;   { UND-Verknüpfung }

     CMD_LIGHT_Lev1 =   $18;   { volle Helligkeit }
     CMD_LIGHT_Lev2 =   $19;   { 87,5% Helligkeit }
     CMD_LIGHT_Lev3 =   $1A;   { 75%   Helligkeit }
     CMD_LIGHT_Lev4 =   $1B;   { 62,5% Helligkeit }
     CMD_LIGHT_Lev5 =   $1C;   { inoffiziell }
     CMD_LIGHT_Lev6 =   $1D;   { inoffiziell }
     CMD_LIGHT_Lev7 =   $1E;   { inoffiziell }
     CMD_LIGHT_Lev8 =   $1F;   { inoffiziell }
```

Weitere globale Konstanten:

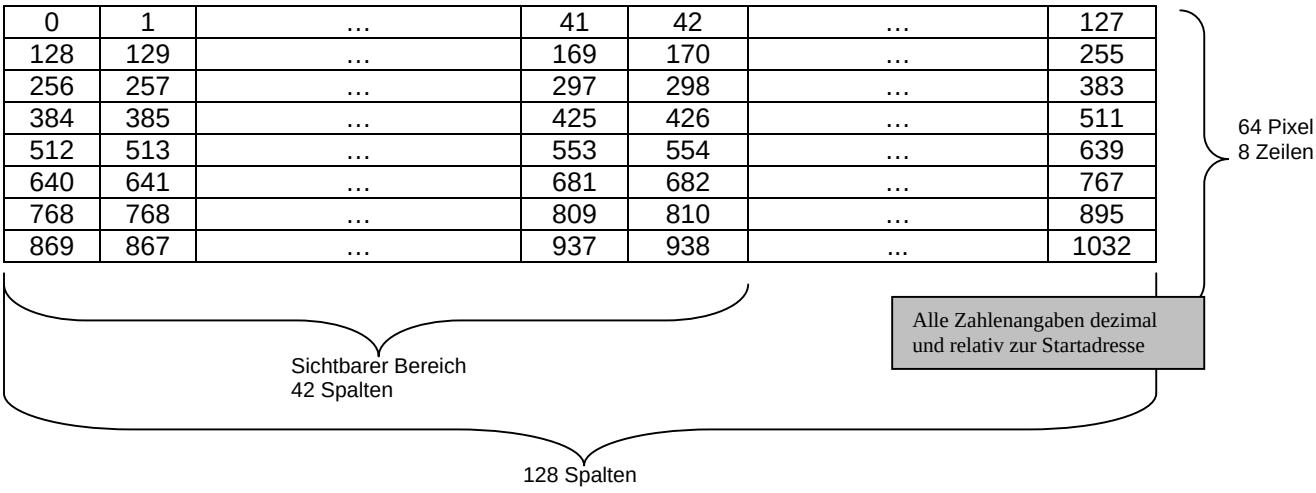
```
{ Displaydimensionen }
     TEXT_SCREEN_WIDTH = 128;
     TEXT_SCREEN_HEIGHT = 8;
     GRAPHICS_SCREEN_WIDTH = 256;
     GRAPHICS_SCREEN_HEIGHT = 64;

{ Adressbereiche SCR1 = $0000, SCR2 = $0800 }
     SCR1_L = $00;      { untere Startaddresse Screen 1 }
     SCR1_H = $00;      { obere Startaddresse Screen 1  }
     SCR2_L = $00;      { untere Startaddresse Screen 2 }
     SCR2_H = $08;      { obere Startaddresse Screen 2  }
```

(Mit eine variablen Startadressen der Screen wäre horizontales und vertikales Scrollen der Diplayanzeige möglich. Auf dieses Feature wurde jedoch verzichtet und die Startadressen als Konstanten definiert.)

Einschub: Displayadressierung

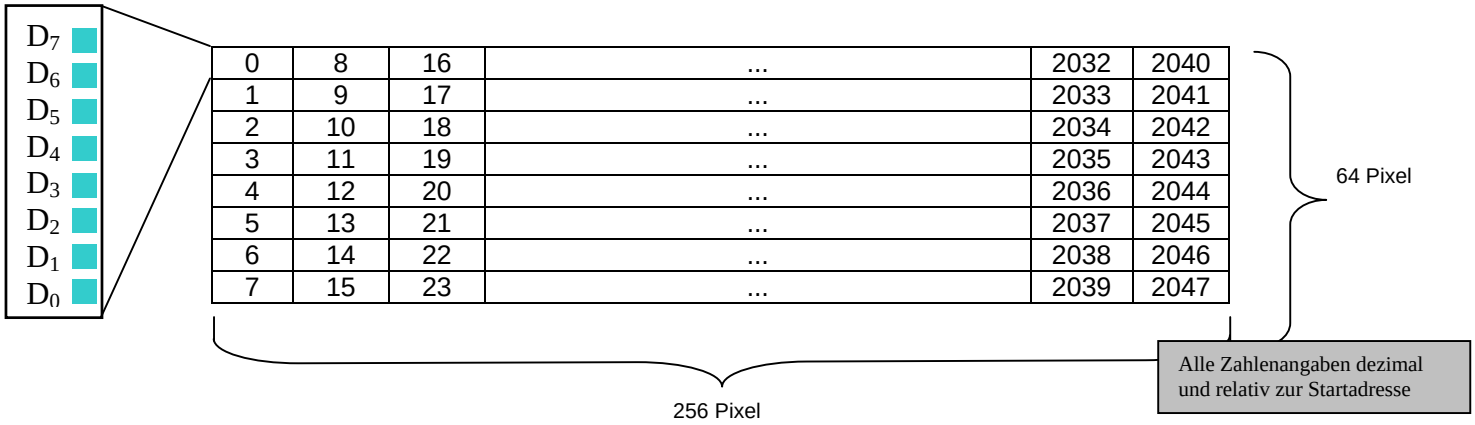
An dieser Stelle ist es nötig die Adressierung des Displays näher zu erklären.
Wie bereits erwähnt hat das Display zwei Screens. Screen1 kann Grafiken darstellen, Screen2 wahlweise Grafiken oder Zeichen aus der internen Charsatz (s. Datenblatt S.5). Beide Screens können auch gleichzeitig angezeigt werden und dabei mit **AND**, **OR** oder **XOR** verknüpft werden. Im VFD-Studio wird Screen2 immer als Charakterscreen betrieben.

Charakterscreen (Screen2):
Ein Charakterscreen hat 1024 Adressen, unterteilt in folgende Adressierung:



Jede Adresse hat eine Größe von 8Bit (1 Byte) und repräsentiert auf 8x6 Pixeln einen Charakter aus dem Display-Charsatz. Das Display hat eine Auflösung von 256x64 Pixeln, woraus sich 8 Zeilen zu 42.7 Zeichen ergeben.
Die obigen Adressangaben sind relativ zur Screenstartadresse. Wenn der Charakterscreen die Startadresse 0800_h (= 2048_{10}) hat, so ist z.B. das zweite Zeichen der fünften Zeile an Adresse $0A01_h$ (= 2561_{10}).

Grafikscreen (Screen1):
Der 256x64 Pixel große Grafikscreen hat insgesamt 16384 Pixel welche in Blöcken zu je 8 Pixeln aufgeteilt sind. Diese 2048 8-bit-Adressen sind folgendermaßen angeordnet:



Das erste Element dieser Adressmap liegt links oben, das nächste darunter. Das 8. Element in der zweiten Spalte ganz oben, usw...
Diese spaltenweise Adressierung zusammen mit der verdrehten Reihenfolge D_7-D_0 ist leider ungünstig, da die üblichen Bildschirmadressierungen im Computerbereich zeilenweise aufgebaut sind und somit einfaches zeilenweises Kopieren aus einer Bildfläche in den Display-RAM nicht ohne Umrechnungen möglich ist.

InitVFD

Diese Prozedur wird unmittelbar nach dem Starten des Hauptfensters von selbigem aufgerufen und setzt die Displaystartadressen für Screen1 und Screen2, sowie den Cursor in autohold-mode (damit er nach Beschreiben einer Adresse nicht zur nächsten weiterläuft).
Die Helligkeit auf höchste Stufe zu setzen ist nicht unbedingt nötig, da dies die Standardeinstellung ist und die Helligkeit ohnehin durch die Script-Befehle in den Listen geändert werden kann.
Die Prozedur Loadchars lädt aus externen Dateien die Zifferndarstellung für die Anzeige der Digitaluhr. Auf sie wird später noch näher eingegangen.

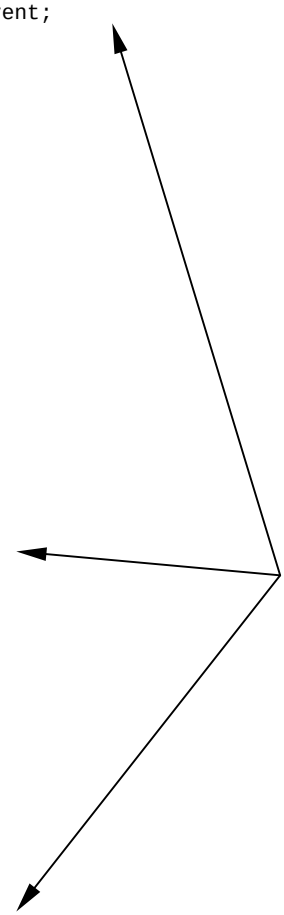
```
procedure TPortIOVFD.InitVFD;       { Display initialisieren }
begin
     // setze Screen1-Startadresse
     WriteCommand(CMD_SCR1_lowADR);
     WriteData(SCR1_L);
     WriteCommand(CMD_SCR1_highADR);
     WriteData(SCR1_H);
     // setze Screen2-Startadresse
     WriteCommand(CMD_SCR2_lowADR);
     WriteData(SCR2_L);
     WriteCommand(CMD_SCR2_highADR);
     WriteData(SCR2_H);
     // setze Cursor in autohold-mode
     WriteCommand(CMD_CUR_HOLD);
     // Volle Helligkeit
     LightLevel:=CMD_LIGHT_Lev1;
     WriteCommand(CMD_LIGHT_Lev1);
     // Charsatz laden
     LoadChars;
end;
```

Wichtig ist die Reihenfolge: erst Screen1, dann Screen2 definieren.
Außerdem: erst das Lower-Byter der Adresse, dann das höherwertige Byte.

Constructor Create (AOwner: TComponent)

Der Constructor erweitert die Create-Methode (ererbt von TComponent) um die Erstellung der Bitmap- und Timerobjekte. Auf deren Funktion wird im einzelnen noch genauer eingegangen. Zu beachten ist hier jedoch die Verknüpfung der OnTimer- und
OnChange-Events zu Prozeduren in dieser Klasse.
So wird z.B. jedes Mal, wenn an dem Bitmapobjekt `Bitmap` Änderungen geschehen die Prozedur `BitmapChangeEvent` ausgeführt.

```
constructor TPortIOVFD.Create (AOwner: TComponent);
begin
     inherited Create (AOwner);
      BA:=$378;                 // Default-Wert beim Laden
     try
      //Animationstimer erstellen
      AnimateTimer:=TTimer.create(self);
      AnimateTimer.enabled:=false;
      AnimateTimer.OnTimer:=AnimateTimerEvent;
     except
      AnimateTimer.free;
     end;
     try
      //AnimationsBitmapObjekt erstellen
      bmp:=TBitmap.create;
     except
      bmp.free;
     end;
     try
      //FadeIn/Out Timer
      FadeTimer:=TTimer.Create(self);
      FadeTimer.Enabled:=false;
     except
      FadeTimer.free;
     end;
     try
      //BitmapObjekt erstellen
      Bitmap:=TBitmap.Create;
      Bitmap.Width:=256;
      Bitmap.Height:=64;
      Bitmap.OnChange:=BitmapChangeEvent;
     except
      Bitmap.free;
     end;
     try
      //ClockBitmapObjekt erstellen
      ClockBitmap:=TBitmap.Create;
      ClockBitmap.width:=64;
      ClockBitmap.height:=64;
     except
      ClockBitmap.Free;
     end;
     try
      //UhrTimer erstellen
      ClockTimer:= TTimer.Create(self);
      ClockTimer.enabled:=false;
      ClockTimer.OnTimer:=ClockTimerEvent;
     except
      ClockTimer.free;
     end;
     try
      //AnimationsBitmapObjekt erstellen
      Bmp:=TBitmap.Create;
     except
      Bmp.free;
     end;
end;
```



Verknüpfung der OnTimer und OnChange Methoden mit eigenen Prozeduren

Destructor Destroy

Alle erstellen Objekte müssen bei Programmende, bzw. beim Freigeben der Komponente, wieder zerstört und der reservierte Speicher freigegeben werden. Dies geschieht im Destructor, der zu diesem Zweck erweitert wurde.
Dabei wird für jedes Objekt geprüft ob es existiert.
Die Anweisungen müssen vor `inherited Destroy;` ausgeführt werden!

```
destructor TPortIOVFD.Destroy;
begin
     if AnimateTimer <> nil then AnimateTimer.Free;
     if bmp <> nil then bmp.free;
     if FadeTimer <> nil then FadeTimer.free;
     if bitmap <> nil then Bitmap.free;
     if ClockBitmap <> nil then ClockBitmap.Free;
     if ClockTimer <> nil then clocktimer.Free;

     inherited Destroy;
end;
```

Register

Die Register-Prozedur registriert die PortIOVFD-Komponente in der Delphi Komponentenbibliothek im Reiter Hardware.

```
procedure Register;
begin
  RegisterComponents('Hardware', [TPortIOVFD]);
end;
```

SetCursor

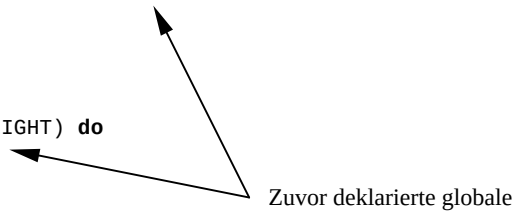
Diese Prozedur dient dazu den Cursor auf eine Adresse von Screen1 oder Screen2 zu setzen.
Sie muss ausgeführt werden, bevor eine Adresse beschrieben werden kann.
Auch beim Cursor muss zunächst das niederwertige Adressbyte geschrieben werden, dann das höherwertige.

```
procedure TPortIOVFD.SetCursor(C_low,C_high: byte);       { Cursor setzen }
begin
   WriteCommand(CMD_CUR_lowADR);
   WriteData(C_low);
   WriteCommand(CMD_CUR_highADR);
   WriteData(C_high);
end;
```

ClearScreen(ScreenNr: byte)

Diese Prozedur löscht den in `ScreenNr` angegebenen Screen, indem es in jedes Adresselement des betreffenden RAM-Bereichs logisch 0 schreibt.
Dazu wird der Cursor zunächst auf Startadresse des betreffenden Screen gesetzt und der Cursormodus in auto-increment Modus. Dann wird dem Display mitgeteilt, dass eine Schreibaktion stattfindet und abhängig davon ob es sich um den Grafikscreen (Screen1) oder den Charakterscreen (Screen2) handelt, 2048 oder 1024 Nullen in den Display-RAM geschrieben.

```
procedure TPortIOVFD.ClearScreen(ScreenNr: byte);       { Screen mit "ScreenNr" löschen }
var
   i: integer;
begin
   // Berechne Startadresse
   if ScreenNr=1 then begin
    SetCursor(SCR1_L,SCR1_H);
   end
   else begin
    SetCursor(SCR2_L,SCR2_H);
   end;
   // setze Cursor in autoinc-mode
   WriteCommand(CMD_CUR_INC);
   // Schreibkommando
   WriteCommand(CMD_WRITE_DATA);
   //löschen
   if ScreenNr=1 then begin
     for i:=1 to (GRAPHICS_SCREEN_WIDTH*GRAPHICS_SCREEN_HEIGHT) do
      begin
       WriteData(0);
      end;
    end
    else begin
     for i:=1 to (TEXT_SCREEN_WIDTH*TEXT_SCREEN_HEIGHT) do
      begin
       WriteData(0);
      end;
   end;
   // setze Cursor in autohold-mode
   WriteCommand(CMD_CUR_HOLD);
end;
```



SelectScreen(Screen: byte);

Diese Prozedur aktiviert den im Parameter `Screen` angegebenen Screen.
D.h. der Inhalt dieses Screens (Daten im betreffenden Adressbereich) werden auf dem Display angezeigt.
Ist `Screen` gleich 3 werden Screen1 und Screen2 gleichzeitig angezeigt.
Die Nummer des Screens (1,2 oder 3 für beide) kann direkt als Command ausgegeben werden (vlg. Abschnitt *Kommandos*)

```
procedure TPortIOVFD.SelectScreen(Screen: byte);       { Aktiven Bildschirm wählen (3=beide) }
begin
   if (Screen<=3)and (Screen>=0) then WriteCommand(Screen);
end;
```

PaintString(s: string; col,row: byte);

Die PaintString-Prozedur schreibt die in `s` übergebene Zeichenkette mit dem Displayzeichensatz in den Screen2-Adressraum.
Die Ausgabe erfolgt in Zeile `row` und Spalte `col`. Ist Screen2 aktiviert (*SelectScreen*) erscheint der Text unmittelbar auf dem Display.
Zunächst werden die Parameter `col` und `row` auf sinnvolle Werte getestet.
Da der Adressraum des Charakterscreen (Screen2) maximal 128x8 Zeichen beinhalten kann dürfen `col` und `row` nicht größer sein und werden ggf. entsprechend beschränkt.
Dann wird bestimmt, ab welcher Adresse des Charakterscreens geschrieben wird.
Diese Adresse wird in der lokalen Variablen position gespeichert, die sich aus der Startadresse des Screen2 und der Textausgabeposition ergibt.
Diese 16bit Adresse wird in zwei 8bit Werte (`a,b`) gespeichert, auf welche der Cursor gesetzt wird. Zudem wird der Cursor noch in den auto-inc Modus gesetzt und an das Display das Schreibkommando gesendet.
Nun kann die Stringübertragung in den Screen2-Adressraum beginnen. Diese erfolgt zeichenweise. Dabei ist zu beachten, dass das Display den ASCII-Wert des Zeichens in eine Zeichendarstellung aus seiner eigenen Charaktertabelle umwandelt und beide Tabellen geringe Unterschiede aufweisen (vgl. Datenblatt S. 5).

```
procedure TPortIOVFD.PaintString(s: string; col,row: byte);       { String an Pos col,row auf Char-
Display ausgeben }
var
   a,b: byte;                    // bilden Adresse für Cursorposition (a=CUR_low, b=CUR_high)
   position: word;               // Adresse als Word
   c: char;                      // Char
   i: integer;                   // for-Variable
begin
     // Überprüfen ob Parameter sinnvoll sind
     if col>(TEXT_SCREEN_WIDTH-1) then col:=(TEXT_SCREEN_WIDTH-1);
     if row>(TEXT_SCREEN_HEIGHT-1) then row:=(TEXT_SCREEN_HEIGHT-1);

     // Adresse berechnen
     position:=SCR2_L+(SCR2_H*$100);                    // Adresse des SCR2 als Offset...
     position:=position+ (row*TEXT_SCREEN_WIDTH)+col;   // ... + Position auf dem Display

     // ermittle CUR_low und CUR_high aus position
     a:=trunc(position mod $100);
     b:=trunc(position / $100);

     // setze Cursor in autoinc-mode
     WriteCommand(CMD_CUR_INC);

     // setze Cursor auf position
     SetCursor(a,b);

     // Schreibkommando
     WriteCommand(CMD_WRITE_DATA);

     // For-Schleife für Zeichenweise Ausgabe des Strings
     for i:= 1 to length(s) do begin
      // kopiere Char aus String s
      c:=s[i];
      // c auf Display ausgeben
      WriteData(ord(c));
     end;
     // setze Cursor in autohold-mode
     WriteCommand(CMD_CUR_HOLD);
end;
```

SetPixelbyte(Pixelbyte,x,y: byte);

Ähnlich wie PaintString arbeitet die SetPixelbyte-Prozedur.
Diese Prozedur zeichnet einen 8 Pixel hohen Balken an die mit `x` und `y` angegebene Position des Grafikscreens.
Es ist leider nicht möglich ein einzelnes Pixel auf das Display zu zeichnen, sondern immer nur 8 zusammenhängende Pixel, die daher im folgenden immer „Pixelbyte“ genannt werden (vlg. *Grafikscreen*)
Dieses Pixelbyte besteht also aus 8 vertikal angeordneten Pixeln und wird durch den Parameter `Pixelbyte` übergeben.
Das oberste Pixel hat dabei die höchste Wertigkeit (2^7), das unterste Pixel die niedrigste (2^0).
Wie in PaintString werden auch bei dieser Prozedur überprüft ob die Positionsparameter sinnvoll sind und ggf. angepasst.
Die Position für den Cursor berechnet sich durch die Startadresse des Grafikscreens und den Positionsangaben `x` und `y` und wird auf zwei 8bit Werte aufgeteilt.
Danach erfolgt die Datenschreibaktion an der betreffenden Adresse.

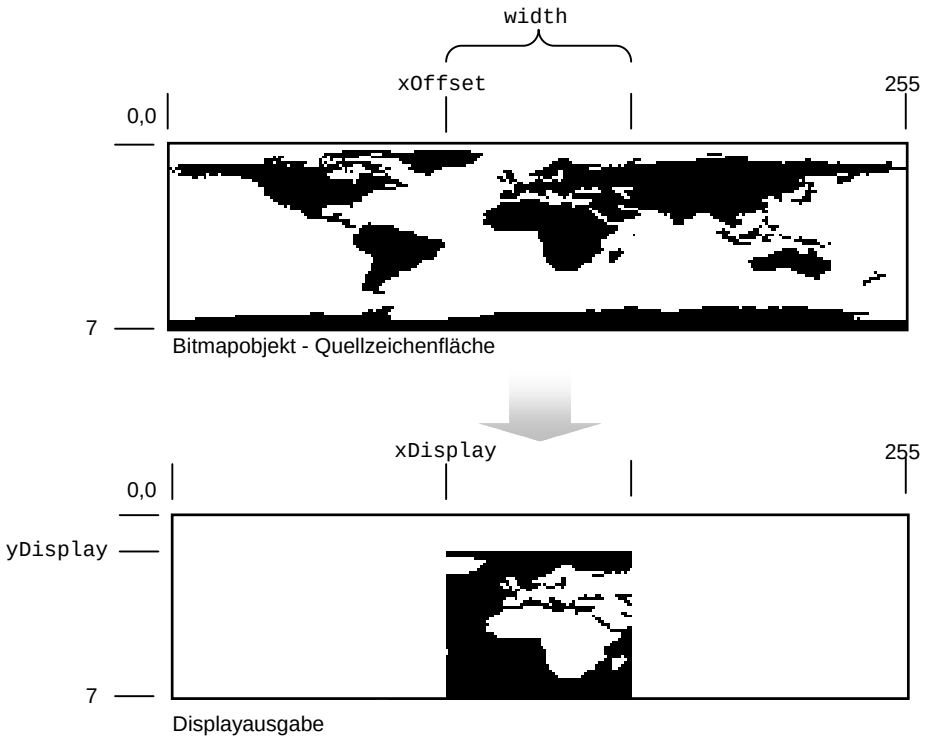
```
procedure TPortIOVFD.SetPixelbyte(PixelByte,x,y: byte); { Pixelbyte auf Display an x,y malen }
var
   a,b: byte;                    // bilden Adresse für Cursorposition (a=CUR_low, b=CUR_high)
   position: word;               // Adresse als Word
begin
     // Überprüfen ob Parameter sinnvoll sind
     if x>(GRAPHICS_SCREEN_WIDTH-1) then x:=(GRAPHICS_SCREEN_WIDTH-1);
     if y>(GRAPHICS_SCREEN_HEIGHT-1) then y:=(GRAPHICS_SCREEN_HEIGHT-1);

     // Adresse berechnen
     position:=SCR1_L+(SCR1_H*$100);                    // Adresse des SCR1 als Offset...
     position:=position+ (x*8)+y;                       // ... + Position auf dem Display

     // ermittle CUR_low und CUR_high aus position
     a:=trunc(position mod $100);
     b:=trunc(position / $100);

     // setze Cursor auf position
     SetCursor(a,b);

     // Schreibkommando
     WriteCommand(CMD_WRITE_DATA);

     // sende Pixelbyte
     WriteData(PixelByte);

     // setze Cursor in autohold-mode
     WriteCommand(CMD_CUR_HOLD);
end;
```

Grafikausgabe auf dem Display

Auf der zuletzt beschriebenen SetPixelbyte-Prozedur basieren sämtliche Grafikausgaben auf dem Display zur Darstellung von Bildern und Animationen.
Die Darstellung von Bildern erfolgt über die Prozedur PaintBitmapRect.

PaintBitmapRect

Diese Prozedur erwartet als Parameter `Source` eine Quellzeichenfläche vom Typ TBitmap, die Ausgabeposition auf dem Display, gegeben durch `x` und `y`, die Breite der Ausgabefläche `witdh` sowie die X-Startposition im der Bitmapzeichenfläche `xOffset` (siehe auch Skizze) und der Malfarbe `Farbe`.
Zur besseren Erklärung der Parameter hier diese Grafische Darstellung:



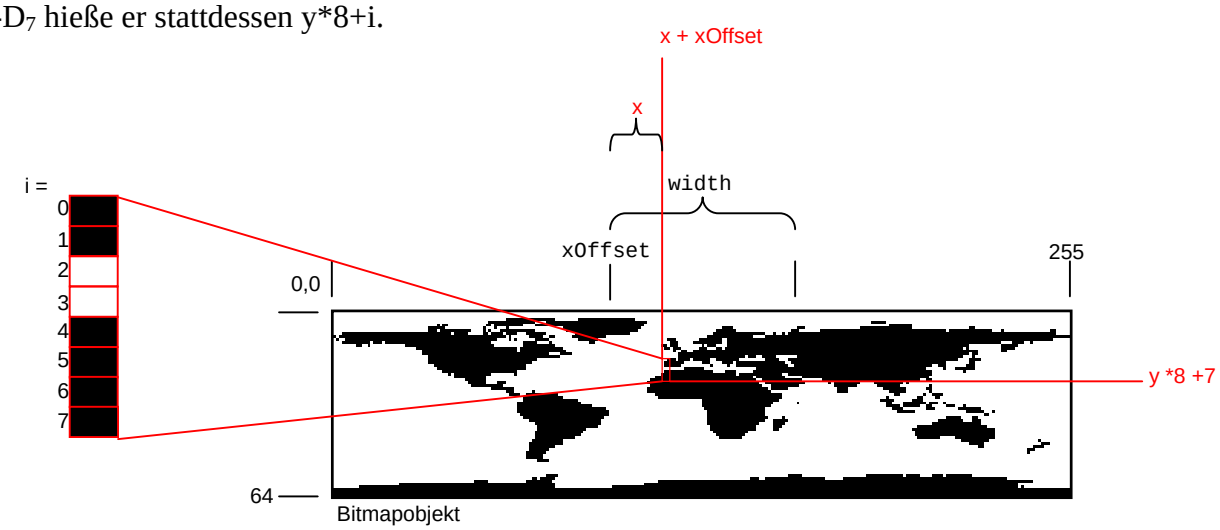
In diesem Fall hätte die Malfarbe `Farbe` den Wert `clWhite` (Weiß).

Mit diesem Parametern ist es möglich einen beliebigen Bitmapbereich auf dem Display anzuzeigen. Es gelten jedoch folgende Einschränkungen:

- Die Größe des Bitmapobjektes, bzw. die seiner Zeichenfläche darf nicht größer als die Displayauflösung sein (256x64 Pixel), sonst wird das anzuzeigende Bild abgeschnitten da diese Methode kein Stretching kennt.
- width und xOffset dürfen einzeln oder zusammenaddiert nicht größer sein als die Breite des Bitmapobjektes (Source.canvas.width)
- Die y-Position auf dem Display kann bedingt durch die Zusammenfassung von je acht Pixeln zu einem vert. Balken nur in Schritten von 8 Pixeln angegeben werden (0..7)

Zuerst wird der Cursor in den AutoIncrement-Modus gesetzt, um ihn bei den Schreiboperationen nicht ständig mit SetCursor auf die entsprechende Adresse zu setzen. Daraufhin wird das Schreibkommando gesendet und das Display erwartet nun Daten.
Diese werden durch drei verschachtelte for-Schleifen an die richtige Adresse geschickt.
Die unterste der Schleifen bildet aus acht übereinanderliegenden Pixeln der Quellbitmap ein Byte. Da nur Schwarzweiß-Grafik dargestellt werden kann (Display kann nur eine Farbe darstellen) ist ein Byte für 8 Pixel ausreichend.
Die Vorgehensweise ist dabei folgende:

In der untersten Schleife (Scheifenvariable i) wird geprüft, ob das Bitmap-Pixel an der Position x+xOffset, y*8+7 –i die Malfarbe `Farbe` hat. Trifft dies zu, so wird zu der Variablen `pixbyte`, die vor jedem Schleifendurchlauf resetet wird, der Wert 2^i addiert.
Zur Veranschaulichung unten noch mal das Bild mit der Weltkarte. Beispielsweise könnte die Position des „Pixelbytes“ gerade bei Gribaltar liegen und die Pixel wie links dargestellt koloriert sein. Nach einem Schleifendurchlauf hätte `pixbyte` dann den Wert 11001111_2 falls als `Farbe` Schwarz gewählt wurde. Wäre dieser Parameter gleich Weiß gewesen, so wäre der Wert gleich 00110000_2 und im Falle, dass die Malfarbe gar nicht gefunden worden wäre eben 00000000_2.
Der Ausdruck y*8+7 –i für die Berechnung der y-Position in der Bitmap ergibt sich übrigens aus der verdrehten Wertigkeit D_7-D_0 der Pixel (siehe Abschnitt *Grafikscreen*) – bei normaler Reihenfolge D_0-D_7 hieße er stattdessen y*8+i.



Ist die pixbyte-Variable anhand der Pixel im Bitmap auf den entsprechenden Wert gesetzt kann die nächsthöhere Schleife (Variable y) mit SetPixelbyte den Wert an das Display schicken.
Bei der Adressierung ist hierbei nun die Ausgabeposition (xDisplay, yDisplay) anzugeben.
Zudem kann der Startadressenoffset entfallen, da Grafikausgabe in diesem Projekt ausschließlich auf Screen1 stattfindet und dieser Screen ab Adresse 0 anfängt.
Wird eine andere Startadresse für die Grafikausgabe verwendet müsste diese dazugerechnet werden.

Zudem sorgt diese zweite Schleife dafür, dass die pixbyte-Variable nach jeder Datenausgabe wieder auf 0 zurückgesetzt wird.

Die oberste Schleife (Variable x) schließlich geht einfach von 0 bis `width` und ruft dabei lediglich die zweite Schleife auf.

PaintBitmapRect

```
procedure TPortIOVFD.PaintBitmapRect(Source: TBitmap;xDisplay,yDisplay,width,xOffset: byte; Farbe:
TColor);       { Zeichenfläche auf Display an Pos x,y mit Breite width ausgeben }
var
   x,y,i: word;                    // Zählervariablen
   pixbyte: byte;                  // Pixelbyte
begin
  // Ausgabe
  for x:=0 to width do begin
   for y:=0 to 7-yDisplay do begin
    // reset pixbyte
    pixbyte:=0;
    //pixbyte erstellen
    for i:=0 to 7 do begin
     if Source.canvas.Pixels[x+xOffset,(y*8)+7-i]=Farbe then pixbyte:=pixbyte+trunc(Power(2,i));
    end;
    SetPixelByte(pixbyte,x+xDisplay,y+yDisplay);
   end;
  end;
  // setze Cursor in autohold-mode
  WriteCommand(CMD_CUR_HOLD);
end;
```

Eine weitere Möglichkeit Bilder zu zeichnen besteht darin, auf der Zeichenfläche des Bitmapobjektes der Klasse zu zeichnen.
Die Klasse PortIOVFD hat eine Variable Bitmap vom Typ TBitmap. Im Konstruktor wird dieses Objekt erstellt

```
     try
      //BitmapObjekt erstellen
      Bitmap:=TBitmap.Create;
      Bitmap.Width:=256;
      Bitmap.Height:=64;
      Bitmap.OnChange:=BitmapChangeEvent;
     except
      Bitmap.free;
     end;
```

und kann dann über <PortIOVFDKomponentenname>.Bitmap angesprochen werden.
Die Bitmap hat eine Größe, die der Auflösung des Displays entspricht. Das Wichtigste ist jedoch die Verknüpfung des OnChange-Events mit der PortIOVFD-Prozedur BitmapChangeEvent.

BitmapChangeEvent

```
procedure TPortIOVFD.BitmapChangeEvent(Sender: TObject);  // RefreshProzedurfür BitmapObjekt->Display
var
   x,y,i: word;                    // Zählervariablen
   pixbyte: byte;                  // Pixelbyte
begin
     // Ausgabe
     for x:=0 to 255 do begin
     for y:=0 to 7 do begin
       // reset pixbyte
       pixbyte:=0;
       //pixbyte erstellen
       for i:=0 to 7 do begin
        if bitmap.canvas.Pixels[x,(y*8)+7-i]=clBlack then pixbyte:=pixbyte+ trunc(Power(2,i));
       end;
       SetPixelByte(pixbyte,x,y);
     end;
     end;
     // setze Cursor in autohold-mode
     WriteCommand(CMD_CUR_HOLD);
end;
```

Die BitmapChangeEvent-Prozedur ist genauso wie die zuvor vorgestellte PaintBitmapRect-Prozedur aufgebaut, mit dem Unterschied, dass keine Parameter berücksichtigt werden müssen.
Die Malfarbe ist immer Schwarz und da das OnChangeEvent keine Auskunft über den geänderten Bereich in der Bitmap gibt muss auch immer der ganze Bereich (256x64 Pixel) ausgegeben werden. Insofern ist diese Prozedur eher ungeeignet für die Ausgabe kleiner Grafiken.

Allerdings bietet diese Prozedur zusammen mit dem Bitmapobjekt nun die Möglichkeit Canvas-Methoden direkt auf dem Display darzustellen:
Die Klasse TCanvas stellt eine abstrakte Zeichenfläche für grafische Objekte dar.
Sie stellt Eigenschaften und Methoden zur Verfügung, wie

- Pinseleinstellungen und Schriftarten festlegen
- Linien und Formen darstellen
- Grafiken zeichnen
- Textausgabe (in versch. Schriftarten)

Und da auch Bitmapobjekte eine Canvaseigenschaft haben, deren Ausgabe auf der Bitmap durch obige Prozedur auf das Display weitergeleitet ist, ist es so möglich die Canvas-Methoden direkt auf dem Display darzustellen.
Beispielweise würde

```
Bitmap.Canvas.MoveTo(0,0);
Bitmap.Canvas.LineTo(253,63);
```

eine diagonale Linie auf dem Display zeichnen.
Oder

```
Bitmap.Canvas.TextOut(32,128,'Hello World!');
```

einen Text unter Verwendung der eingestellten Schriftart an gewählter Position ausgeben.

Sämtliche Grafikmethoden der Canvas-Klasse sind also auch auf das Display anwendbar. Allerdings gilt zu beachten, dass nach jeder Bitmapänderung die Bitmap komplett neu auf dem Display gezeichnet wird und dieser Vorgang relativ langsam und rechenintensiv ist.
Aus genau diesem Grund wurde die Darstellung sich ständig ändernder Systeminformationen (Beispiel Uhrzeit) in Schriftart, also als Grafikausgabe, nicht berücksichtigt.

Animationen

Ein weiterer Punkt der Grafikausgabe ist das Abspielen von Animationen.
Zunächst waren dazu ein paar Vorüberlegungen sinnvoll.

- Die einzelnen Animationsframes dürften maximal 256x64 Pixel groß sein
- Das Zeichnen eines Einzelbildes dieser Größe erfordert jedoch bereits so viel Rechenkapazität, dass Animationen eher sehr klein gehalten werden sollten und außerdem
- keine hohe Framerate zu erwarten sein sollte. Die Abspielgeschwindigkeit sollte aber auf jeden Fall einstellbar sein.

Da die Animationen letztlich auch aus irgendeiner Dateiquelle stammen müssen standen zudem Überlegungen bezüglich des Dateiformates an.
Eine Möglichkeit wäre Animationen aus AVI-Filmen abzuspielen. Dagegen spricht jedoch, dass AVI nur ein Containerformat für eine Vielzahl unterschiedlicher Videocodecs ist und das Programm entsprechend viele Formate unterstützen sollte. Zudem hat nicht jeder Benutzer die Möglichkeit AVIs selbst zu erstellen.
Ein anderes Animationsformat sind GIF- und PNG-Dateien. Für beide Formate gibt es eine Vielzahl von Programmen zur Erstellung eigener Anmationen. Allerdings fand sich bei der Recharge über den Aufbau eines Gif- oder Png-Bildes zu wenig Informationsmaterial.
Daher wurde schließlich das Format benutzt, welches ohnehin schon implementiert war – Bitmap.
Eine Bitmapdatei kann durchaus zur Speicherung von Animationen benutzt werden, wie untenstehendes Bild zeigt.

Diese Bitmap-Datei stellt die Animation der Erdkugel dar. Die einzelnen Frames sind alle gleich groß und liegen horizontal nebeneinander.

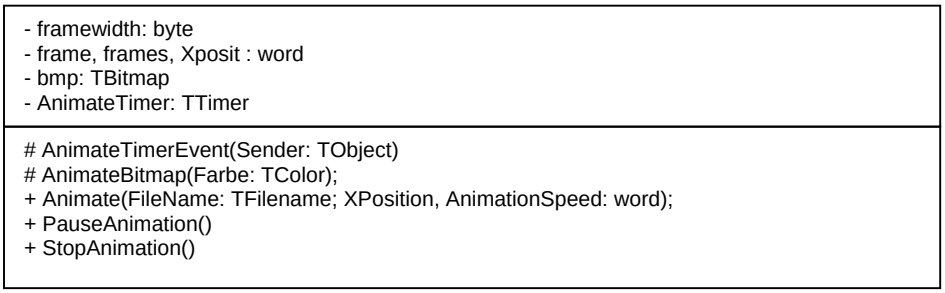
Folgende Vorgaben wurden festgelegt:
- Die Bitmaphöhe und somit auch die Höhe der Einzelbilder muss 64 Pixel betragen.
- Alle Einzelframes müssen die gleiche Breite haben und dürfen nicht breiter als 256 Pixel sein.
- Die Breite der Bitmap geteilt durch die Framebreite ergibt somit die Anzahl der Bilder, bzw. die Bitmapbreite durch Bilderanzahl ergibt die Framebreite.

Einziges Problem dabei ist, dass das Bitmapformat nicht noch zusätzliche Informationen speichern kann, die Auskunft über Framebreite oder Anzahl der Bilder liefern.
Dieses Problem wurde schlicht und einfach durch den Dateinamen gelöst: jedes Animationsbild muss in den ersten zwei Zeichen des Dateinamens die Anzahl der Bilder enthalten.
Obiges Bild hat z.B. den Dateinamen „30globus.bmp“, woraus sofort hervorgeht, dass es sich um 30 Einzelframes handelt. Und da die Bitmap 1920 Pixel breit ist hat jedes Frame eine Breite von 1920/30 = 64 Pixeln.

Als Konsequenz dieser Definition kann eine Animation maximal 99 Einzelbilder haben.
Und ein Bild mit weniger als zehn Frames müsste z.B. „05Animation.bmp“ heißen.

Diesen Einschränkungen sind jedoch akzeptabel, zumal so eigene Animationen selbst mit primitivsten Malprogrammen hergestellt werden können.

An der Darstellung der Animationen auf dem Display sind folgende Prozeduren, Variablen und Objekte beteiligt:



- *framewidt* enthält die Breite der Einzelbilder
- *frame* gibt das aktuelle Frame an, frames die Gesamtanzahl Einzelbilder
- *Xposit* bezeichnet die Ausgabeposition der Animation auf dem Display
- *Bmp* wird benutzt um die Animationsdatei zu laden
- *AnimateTimer* kümmert sich um FrameRate und Animationsausgabe
- *AnimateTimerEvent* ist mit dem OnTimer-Event des AnimateTimers verknüpft
- *AnimateBitmap* zeichnet das aktuelle Frame auf das Display
- *Animate* spielt die Animation ab
- *PauseAnimation* hält den AnimateTimer an
- *StopAnimation* beendet die Animation

Da sie am wenigsten Erklärungsbedarf mit sich bringen, zunächst die beiden public Prozeduren *PauseAnimation* und *StopAnimation*.
Der Name sagt bereits welchem Zweck sie dienen und der Code dürfte auch ohne weitere Erklärungen verständlich sein.

```
procedure TPortIOVFD.PauseAnimation;
begin
     if bmp<>nil then AnimateTimer.enabled:=not AnimateTimer.enabled;
     //bmp=nil würde zu Exception führen
end;

procedure TPortIOVFD.StopAnimation;
begin
     AnimateTimer.enabled:=false;
     frame:=0;
end;
```

Die Prozedur StopAnimation wird automatisch aufgerufen, sobald ein neuer Screen aufgebaut wird (d.h. sobald andere Informationen dargestellt werden sollen).

Neben diesen beiden gibt es noch eine weitere public Prozedur *Animate*, die den Timer aktiviert, die Frameanzahl aus dem Dateinamen der Animationsdatei ermittelt sowie die Animationsgeschwindigkeit einstellt.

```
procedure TPortIOVFD.Animate(FileName: TFilename; XPosition, AnimationSpeed: word);
begin
   //ermittle Anzahl der Frames aus Dateiname
   frames:=strtoint(copy(extractfilename(FileName),0,2));
   bmp.LoadFromFile(FileName); //Datei laden
   framewidth:=bmp.width div frames;  //Framebreite berechnen
   selectscreen(3);   // beide Screens an
   XPosit:=XPosition;
   AnimateTimer.Enabled:=false;
   AnimateTimer.Interval:=AnimationSpeed;
   AnimateTimer.Enabled:=true;
end;
```

Sie aktiviert den AnimateTimer, dessen OnTimer-Event mit der Prozedur AnimateTimerEvent verknüpft ist:

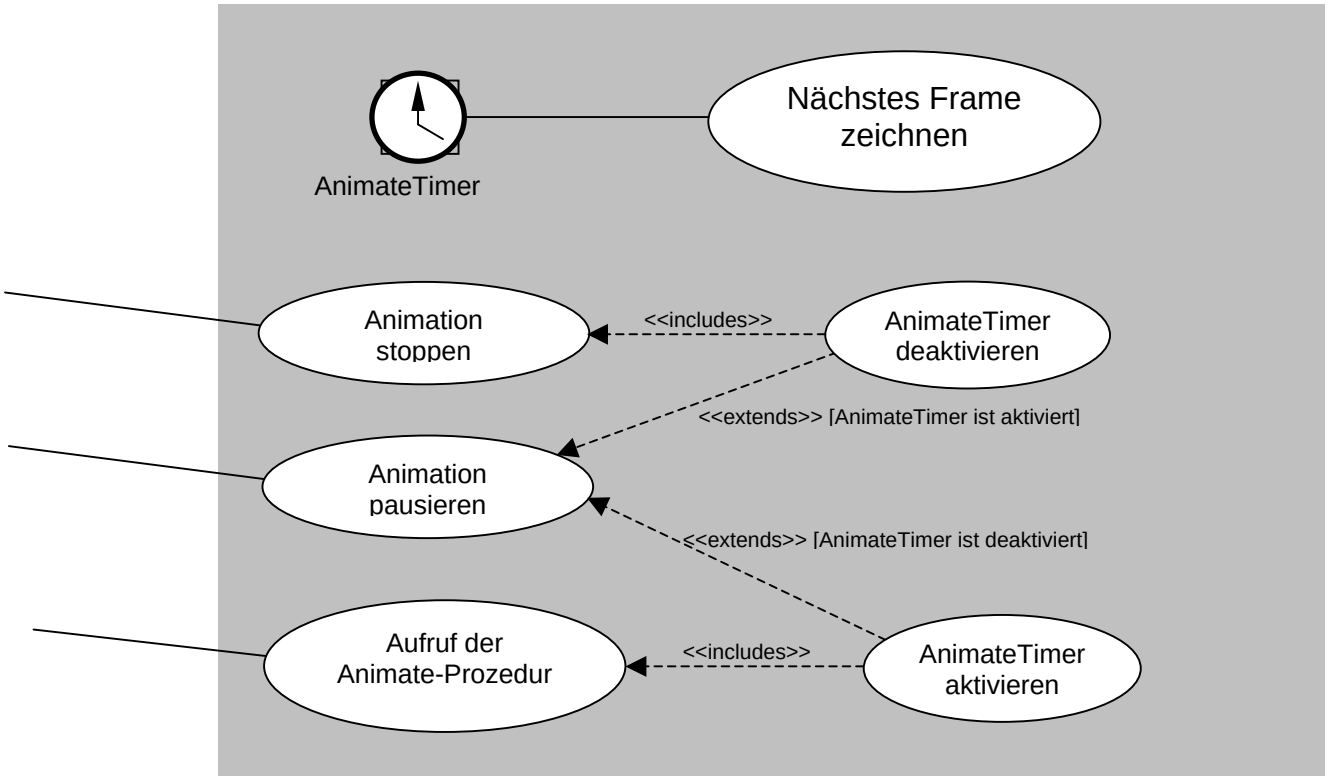
```
procedure TPortIOVFD.AnimateTimerEvent(Sender: TObject);
begin
     AnimateBitmap(clBlack);
     inc(frame);
     if frame >= frames then frame:=0;
end;
```

Der Timer also inkrementiert die Variable `frame`, welche das aktuelle Einzelbild der Animation bezeichnet und ruft `AnimateBitmap` auf.

AnimateBitmap zeichnet anhand der Variable `frame` das zugehörige Einzelbild der Animation aus bmp auf das Display. Die Arbeitsweise ist vergleichbar mit der Prozedur PaintBitmapRect, weshalb nicht nochmals näher darauf eingegangen werden muss.

```
procedure TPortIOVFD.AnimateBitmap(Farbe: TColor);
var
   x,y,i: word;                    // Zählervariablen
   pixbyte: byte;                  // Pixelbyte
begin
     // Schreibkommando
     WriteCommand(CMD_WRITE_DATA);
     // Ausgabe
     for x:=0 to Framewidth do begin
     for y:=0 to 7 do begin
       // reset pixbyte
       pixbyte:=0;
       //pixbyte erstellen
       for i:=0 to 7 do begin
        if bmp.canvas.Pixels[x+Frame*Framewidth,(y*8)+7-i]=Farbe then pixbyte:=pixbyte+ trunc(Power(2,i));
       end;
       SetPixelByte(pixbyte,x+xposit,y);
     end;
     end;
     // setze Cursor in autohold-mode
     WriteCommand(CMD_CUR_HOLD);
end;
```

Abschließend noch einmal eine stark vereinfachte Übersicht über das Abspielen von Animationen anhand eines Use-Case:



Uhrzeitdarstellung

Die Uhrzeit kann von der PortIOVFD-Klasse wahlweise analog oder digital dargestellt werden (es gibt natürlich auch noch die Möglichkeit die Uhrzeit als String über die Prozedur PaintString auszugeben).

Bei der analogen Darstellung wird ein 64x64 Pixel großes Ziffernblatt mit Zeigern für Stunden, Minuten und Sekunden auf das Display gezeichnet. Die Position auf dem Display kann angegeben werden.
Zu beachten ist hier, dass die Darstellung jede Sekunde aktualisiert wird und dabei das Ziffernblatt jedes Mal neu gezeichnet wird. Diese Darstellung benötigt also vergleichbar viel Systemressourcen wie eine 64x64 Pixel große Animation mit 1 Frame/Sek.


Analoguhr

Die Digitaluhr wird als Schriftzug auf das Display gezeichnet, jedoch nicht mit dem Display-Zeichensatz sondern mit extern erstellten Zeichensätzen, welche in den Dateien *0.dat .. 9.dat* vorliegen. Diese Dateien enthalten, wie der Name bereits ahnen lässt, den Zeichensatz für die jeweilige Ziffer 0..9.
Jede Datei enthält die Daten für die Darstellung einer 32x48 Pixel großen Ziffer.
Die Dateien sind bereits für die Darstellung auf dem Display optimiert, d.h. sie sind nicht wie eine gewöhnliche Bitmap aufgebaut, sondern gemäß der Adressierung des Display RAM. Sie bestehen daher aus 192 Bytes (32x48 / 8bit), die nacheinander eingelesen und auf dem Display ausgegeben werden können.

Ständige Lesezugriffe auf externe Dateien würden jedoch unnötig viele Systemressourcen benötigen und daher werden die .dat-Dateien nur einmal beim Programmstart (genauer: beim Initialisieren durch InitVFD) eingelesen und im Array *DC* im Speicher abgelegt.
Dieses Einlesen des Zeichensatzes in den Speicher übernimmt die Prozedur LoadChars, die von InitVFD aufgerufen wird.

```
procedure TPortIOVFD.LoadChars;
var F: File of Byte;
    i: Word;   // Byteindex
    Fi: byte;  // Fileindex
    b: byte;   // (Pixel-)Byte
begin
  for Fi:=0 to 9 do begin
   AssignFile(F, extractfilepath(application.exename)+ inttostr(Fi)+'.dat');
   Reset(F);
   i:=0;
   while i<192 do begin
    Read(F, b);
    DC[Fi, i]:=b;
    i:=i+1;
   end;
   CloseFile(F);
  end; //Fi
end;
```

Um die Uhrzeit anzuzeigen wird die Prozedur *Time* aufgerufen:

```
procedure TPortIOVFD.Time(analog: boolean; Xposition: byte);
var
   xoff,yoff: Integer;  //wo?
   x,y: Word;  //Koordinaten
   i: word;    //Zähler
   ziffer: byte; //Ziffer
   h,m,s,ms: word;
begin
     if analog = true then begin
      digital:=false;
      xUhr:=xPosition;
     end
     else begin  // digital
      clearscreen(1);    // GFX_Screen löschen
      digital:=true;
      DoppelPunkt;
      RefreshTime(true);
     end;   // end else
     clocktimer.Enabled:= true; // Uhrentimer aktivieren - dessen OnTimerEvent
                                // kümmert sich um Aktualisierung
end;
```

Sie aktiviert den ClockTimer, der von nun an jede Sekunde die aktuelle Uhrzeit ausgibt.
Damit dieser weis, ob die Uhrzeit analog oder digital gezeigt werden soll wird dies in der globalen Variablen `digital` gespeichert.
Soll die Uhr als analoges Ziffernblatt gezeichnet werden, so wird auch die Position auf dem Display in einer globalen Variable hinterlegt, in `xUhr`.
Für die Digitaluhr ist dieser Parameter irrelevant, die Digitaluhr benötigt ohnehin die ganze Zeichenfläche. Daher wird in diesem Fall auch der Grafikscreen gelöscht.
Weiterhin wichtig für die Digitaluhr ist die Prozedur *DoppelPunkt*, die nichts anderes macht als die Doppelpunkte für die digitale Uhr anzuzeigen (12:30:56).

```
procedure TPortIOVFD.DoppelPunkt;       { Zeichnet Doppelpunkte für die Uhr }
begin
     SetPixelbyte(255,175,4);
     SetPixelbyte(255,174,4);
     SetPixelbyte(255,173,4);
     SetPixelbyte(255,172,4);
     SetPixelbyte(255,171,4);
     SetPixelbyte(255,170,4);

     SetPixelbyte(255,175,6);
     SetPixelbyte(255,174,6);
     SetPixelbyte(255,173,6);
     SetPixelbyte(255,172,6);
     SetPixelbyte(255,171,6);
     SetPixelbyte(255,170,6);


     SetPixelbyte(255,90,4);
     SetPixelbyte(255,89,4);
     SetPixelbyte(255,88,4);
     SetPixelbyte(255,87,4);
     SetPixelbyte(255,86,4);
     SetPixelbyte(255,85,4);

     SetPixelbyte(255,90,6);
     SetPixelbyte(255,89,6);
     SetPixelbyte(255,88,6);
     SetPixelbyte(255,87,6);
     SetPixelbyte(255,86,6);
     SetPixelbyte(255,85,6);

end;
```

Punkt oben rechts

Punkt unten rechts

Punkt oben links

Punkt unten links

Nachdem der ClockTimer aktiviert wurde, ruft er jede Sekunde die Prozedur *ClockTimerEvent* auf, die die Analoguhr zeichnet, oder ihrerseits die Prozedur RefreshTime aufruft (welche die Digitaluhr zeichnet):

```
procedure TPortIOVFD.ClockTimerEvent(Sender: TObject);
const
     strichle = 7;   // Länge der Stundenstriche am Uhrrand
     SekRad= 31;     // Länge des Sekundenzeigers
     MinRad= 20;     // Länge des Minutenzeigers
     HrRad = 15;     // Länge des Stundenzeigers
var
   h,m,s,ms: word;
begin
     if digital = false then begin        // zeigt Analoguhr an
     with ClockBitmap.Canvas do begin     // zeichnen auf Uhrenbitmap
     Pen.Width:=2;
     //Kreis
     Ellipse(0,0,64,64);
     //Strichle
     Pen.Width:=1;
     MoveTo(32,0);
     LineTo(32,strichle);
     MoveTo(32,63);
     LineTo(32,63-strichle);
     MoveTo(0,32);
     LineTo(strichle,32);
     MoveTo(63,32);
     LineTo(63-strichle,32);
     MoveTo(4,17);
     LineTo(12,21);
     MoveTo(17,4);
     LineTo(21,12);
     MoveTo(59,17);
     LineTo(51,21);
     MoveTo(46,4);
     LineTo(42,12);
     MoveTo(4,63-17);
     LineTo(12,63-21);
     MoveTo(17,63-4);
     LineTo(21,63-12);
     MoveTo(59,63-17);
     LineTo(51,63-21);
     MoveTo(46,63-4);
     LineTo(42,63-12);
     //Zeit dekodieren
     decodetime(now,h,m,s,ms);
     if h>12 then h:=h-12;  //von 24h nach 12h - Format
     //Umrechnungen in Grad
     h:=h*6*5;
     m:=m*6;
     s:=s*6;

     Pen.Width:=1;
     moveto(32,32);
     lineto(32+ round(31*sin(DegToRad(s))) , 32-round(31*cos(DegToRad(s))) );

     Pen.Width:=1;
     moveto(32,32);
     lineto(32+ round(SekRad*sin(DegToRad(s))) , 32-round(SekRad*cos(DegToRad(s))) );

     Pen.Width:=2;
     moveto(32,32);
     lineto(32+ round(MinRad*sin(DegToRad(m))) , 32-round(MinRad*cos(DegToRad(m))) );

     Pen.Width:=3;
     moveto(32,32);
     lineto(32+ round(HrRad*sin(DegToRad(h+m/12)) ) , 32-round(HrRad*cos(DegToRad(h+m/12))) );
     end;

     SelectScreen(3);
     PaintBitmapRect(ClockBitmap,xUhr,0,64,0,clblack);
     end // digital = false
     else RefreshTime(false);      // zeigt Uhrzeit digital an

end;
```

Die Digitaluhr wird durch die Prozedur *RefreshTime* gezeichnet.
Würde diese Prozedur jedoch jedes Mal alle Ziffern zeichnen, wäre die CPU-Belastung zu groß. Daher wird jedes Mal nur die Einerstelle der Sekunden aktualisiert. Erst wenn diese von 9 auf 0 wechselt wird auch die Zehnerstelle der Sekunden aktualisiert. Und nur wenn die Sekunden-Zehnerstelle auf 0 wechselt, wird die Minuten-Einerstelle aktualisiert, usw...
Damit aber alle Ziffern am Anfang einmal gezeichnet werden können hat die Prozedur noch einen boolean-Parameter, der bewirkt, dass alle Ziffern unabhängig von der vorherigen gezeichnet werden. Die Prozedur Time nutzt diesen Parameter (s. oben).

```
procedure TPortIOVFD.RefreshTime(PaintAll: boolean);
var
   xoff,yoff: Word;  //wo?
   x,y: Word; //Koordinaten
   i: word;    //Zähler
   ziffer: byte; //Ziffer
   h,m,s,ms: word;
begin
     x:=0;
     y:=0;

     xoff:=220;
     yoff:=2;

     decodetime(now,h,m,s,ms);

     //Sekunden-Einer
     ziffer:=s mod 10;
     for i:=0 to 191 do begin
      setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
      y:=y+1;
      if y=6 then begin
       y:=0;
       x:=x+1;
      end;
     end;

      if (Ziffer = 0) or (PaintAll) then begin
       xoff:=xoff-70;
       //Sekunden-Zehner
       ziffer:=s div 10;
       for i:=0 to 191 do begin
        setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
        y:=y+1;
        if y=6 then begin
         y:=0;
         x:=x+1;
        end;
       end;
      end; //Ziffer =0

      if (Ziffer = 0) or (PaintAll) then begin
       xoff:=xoff-80;
       //Minuten-Einer
       ziffer:=m mod 10;
       for i:=0 to 191 do begin
        setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
        y:=y+1;
        if y=6 then begin
         y:=0;
         x:=x+1;
        end;
       end;
      end; //Ziffer =0

      if (Ziffer = 0) or (PaintAll) then begin
       xoff:=xoff-70;
       //Minuten-Zehner
       ziffer:=m div 10;
       for i:=0 to 191 do begin
        setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
        y:=y+1;
        if y=6 then begin
         y:=0;
         x:=x+1;
```

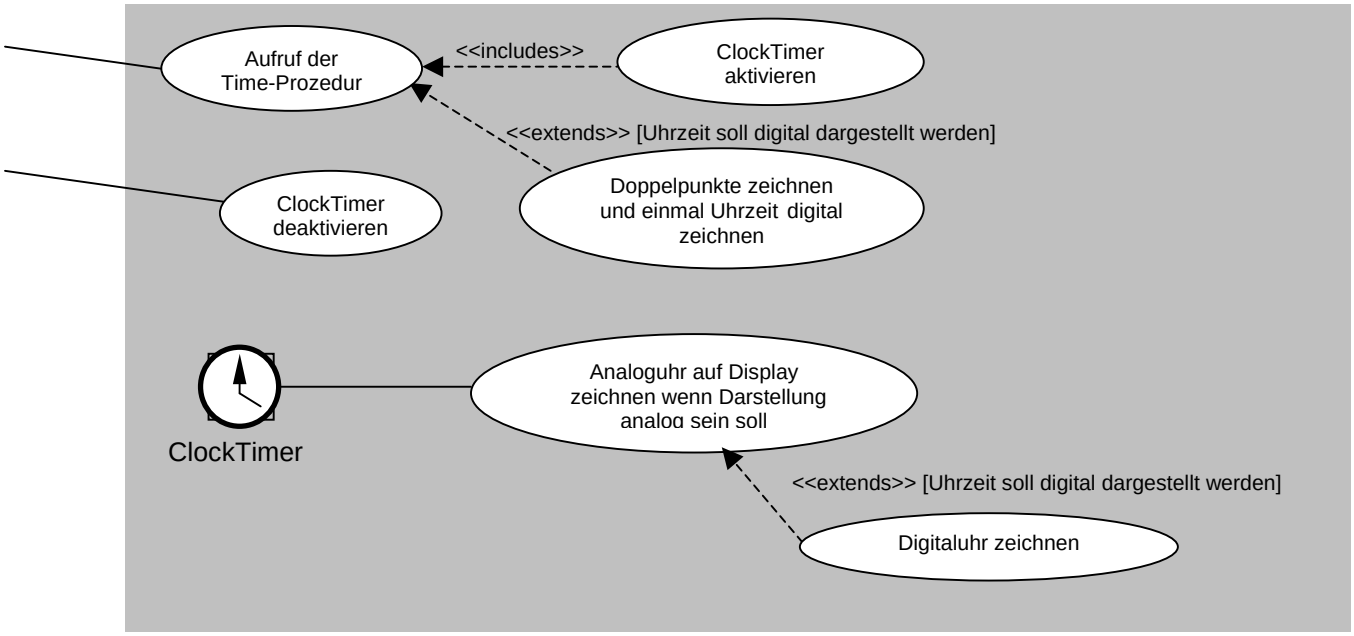
```
        end;
       end;
      end; //Ziffer =0

      if (Ziffer = 0) or (PaintAll) then begin
       xoff:=xoff-80;
       //Stunden-Einer
       ziffer:=h mod 10;
       for i:=0 to 191 do begin
        setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
        y:=y+1;
        if y=6 then begin
         y:=0;
         x:=x+1;
        end;
       end;
      end; //Ziffer =0

      if (Ziffer = 0) or (PaintAll) then begin
       xoff:=xoff-70;
       //Stunden-Zehner
       ziffer:=h div 10;
       for i:=0 to 191 do begin
        setpixelbyte(DC[ziffer,i],xoff+x,yoff+y);
        y:=y+1;
        if y=6 then begin
         y:=0;
         x:=x+1;
        end;
       end;
      end; //Ziffer =0
end;
```

Schließlich fehlt noch eine Prozedur um die Uhrzeitausgabe wieder zu beenden, damit die folgenden Screens nicht von der Uhrzeit überschrieben werden.
Dies geschieht am einfachsten dadurch, den ClockTimer zu deaktivieren:

```
procedure TPortIOVFD.stopClock;
begin
     clocktimer.Enabled:= false;
end;
```

Die StopClock-Prozedur wird jedes Mal aufgerufen, wenn das Hauptprogramm einen neuen Screen aufbaut.

Abschließend noch einmal eine vereinfachtes Use-Case zur Illustration der Uhrzeitdarstellung:



Ein- und Ausblendeffekte

Das Display unterstützt laut Datenblatt vier Helligkeitsstufen.
Eigene Versuche ergaben jedoch , dass es neben diesen noch vier weitere gibt.
Damit lässt sich ein relativ sanfter Ein- bzw. Ausblendeffekt steuern.
Aufgerufen wird dieser durch die Prozedur *Fade*:

```
procedure TPortIOVFD.Fade(off: boolean; Speed: word);
begin
      selectscreen(3); // beide Screens an
      FadeTimer.Enabled:=false;
      FadeTimer.Interval:=Speed;
      FadeTimer.OnTimer:=FadeTimerEvent;
      Fout:= off;
      FadeTimer.Enabled:=true;
end;
```

Diese speichert die Überblendrichtung (Einblenden oder Ausblenden) in der globalen Variable *Fout* und aktiviert den FadeTimer, dessen Intervall die Geschwindigkeit des Ein- oder Ausblendens bestimmt.

Dem OnTimer-Event des FadeTimer ist die Prozedur *FadeTimerEvent* zugeordnet:

```
procedure TPortIOVFD.FadeTimerEvent(Sender: TObject);
begin
     if Fout= true then LightLevel:=LightLevel+1
     else LightLevel:=LightLevel-1;
     if (LightLevel> CMD_LIGHT_Lev8) then begin
      LightLevel:=CMD_Light_Lev8;
      FadeTimer.enabled:=false;
     end;
     if (LightLevel< CMD_LIGHT_Lev1) then begin
      LightLevel:=CMD_Light_Lev1;
      FadeTimer.enabled:=false;
     end;
     WriteCommand(LightLevel);
end;
```

Abhängig von *Fout* inkrementiert oder dekrementiert sie die globale Variable *LightLevel*, welche den aktuellen Helligkeitswert speichert und sendet diesen Helligkeitswert an das Display.
Hat *LightLevel* das obere oder untere Ende der Helligkeitsskala erreicht, wird der FadeTimer deaktiviert – das Ein-/Ausblenden ist beendet.