

```

{ ****
*                               S 6 4 3 5 P . P A S                               *
**-----**
* Task                          : Demonstrates the work with sprites in the      *
*                               640x350 EGA and VGA graphic modes, using 16      *
*                               colors and two screen pages. This program      *
*                               requires assembler routines from modules      *
*                               V16COLPA.ASM and S6435PA.ASM.                  *
*                               ****                                           *
**-----**
* Author                        : MICHAEL TISCHER                               *
* Developed on                  : 12/05/90                                     *
* Last update                   : 01/04/91                                     *
*****}

```

```
program S6435P;
```

```
uses dos, crt;
```

```
{-- External references to the assembler routines -----}
```

```
{ $L v16colpa } { Link assembler module }
```

```

procedure init640350; external;
procedure setpix( x, y : integer; pcolor : byte ); external;
function getpix( x, y : integer ) : byte ; external;
procedure setpage( page : integer ); external;
procedure showpage( page : integer ); external;

```

```
{ $L s6435pa } { Link assembler module }
```

```
procedure CopyVideo2Buf( bufptr : pointer;
```

```

        page      : byte;
        fromx,
        fromy     : integer;
        rwidth,
        rheight   : byte      ); external;

procedure CopyBuf2Video( bufptr  : pointer;
                        page     : byte;
                        tox,
                        toy      : integer;
                        rwidth,
                        rheight  : byte ); external;

procedure MergeAndCopyBuf2Video( spribufptr,
                                hgbufptr,
                                andbufptr  : pointer;
                                page        : byte;
                                tox,
                                toy         : integer;
                                rwidth,
                                rheight    : byte ); external;

{-- Constants -----}

const MAXX = 639;           { Maximum X- and Y-coordinates }
      MAXY = 349;

      OUT_LEFT   = 1;      { For collision documentation in SpriteMove() }
      OUT_TOP    = 2;
      OUT_RIGHT  = 4;
      OUT_BOTTOM = 8;
      OUT_NO     = 0;      { None }

```

```

{-- Type declarations -----}

type PIXBUF = record           { Information for GetVideo and PutVideo }
    widthbytes,                { Width of area in bytes }
    numrows : byte;            { Number of rows }
    pixblen: integer;          { Length of pixel buffer }
    pixbptr: pointer;          { Pointer to the pixel buffer }
end;
PIXPTR = ^PIXBUF;              { Pointer to a pixel buffer }

SPLOOK = record                { Sprite design }
    twidth,                    { Total width }
    theight : byte;            { Height in pixel rows }
    bmskp : array [0..7] of pointer; { Ptr to AND buffer }
    pixmp : array [0..7] of PIXPTR; { Ptr to pixel def. }
end;                           { its page was discarded }
SPLP = ^SPLOOK;                { Pointer to design }

SPID = record                  { Sprite descriptor (ID) }
    splookp : SPLP;            { Pointer to design }
    x, y : array [0..1] of integer; { Coordinates: pp.0&1 }
    hgptr : array [0..1] of PIXPTR; { Pointer to back- }
end;                           { ground buffer }
SPIP = ^SPID;                  { Pointer to sprite descriptor }

BYTEAR = array[0..10000] of byte; { Addressing }
BARPTR = ^BYTEAR;                { different buffers }

VCARD = ( EGA, VGA, NEITHERNOR );

{*****}

```

```

*   IsEgaVga : Determines whether EGA or VGA card is installed.           *
**-----**
*   Input    : None                                                         *
*   Output   : EGA, VGA or NEITHERNOR                                     *
*****}

```

```
function IsEgaVga : VCARD;
```

```
var Regs : Registers;           { Processor register for interrupt call }
```

```
begin
```

```
    Regs.AX := $1a00;           { Function 1AH applies to VGA only }
```

```
    Intr( $10, Regs );
```

```
    if ( Regs.AL = $1a ) then           { Function available? }
```

```
        IsEgaVga := VGA
```

```
    else
```

```
        begin
```

```
            Regs.ah := $12;           { Call function 12H, }
```

```
            Regs.bl := $10;           { sub-function 10H }
```

```
            intr($10, Regs);           { Call video BIOS }
```

```
            if ( Regs.bl <> $10 ) then IsEgaVga := EGA
```

```
                else IsEgaVga := NEITHERNOR;
```

```
        end;
```

```
end;
```

```
{*****}
```

```
*   PrintChar : Writes a character to the screen while in graphic mode.*
```

```
**-----**
```

```
*   Input    :   THECHAR = Character to be written                       *
```

```
*               x, y     = X- and Y-coordinates of upper left corner      *
```

```
*               FG       = Foreground color                               *
```

```
*               BK       = Background color                               *
```

```

*   Info       : Character is created in an 8x8 matrix, based on the      *
*               8x8 ROM font.                                             *
*****}

```

```

procedure PrintChar( thechar : char; x, y : integer; fg, bk : byte );

type FDEF = array[0..255,0..7] of byte;           { Font array }
      TPTR = ^FDEF;                               { Pointer to font }

var  Regs   : Registers;                          { Registers for interrupt call }
      ch    : char;                               { Individual pixels in character }
      i, k,                               { Loop counter }
      BMask : byte;                               { Bit mask for character design }

const fptr : TPTR = NIL;                         { Pointer to font in ROM }

begin
  if fptr = NIL then                             { Pointer to font already set? }
    begin                                         { No }
      Regs.AH := $11;                            { Call video BIOS function 11H, }
      Regs.AL := $30;                            { sub-function $30 }
      Regs.BH := 3;                              { Get pointer to 8x8 font }
      intr( $10, Regs );
      fptr := ptr( Regs.ES, Regs.BP );           { Set pointers }
    end;

  if ( bk = 255 ) then                           { Drawing transparent characters? }
    for i := 0 to 7 do                           { Yes --> Set foreground pixels only }
      begin
        BMask := fptr^[ord(thechar),i]; { Get bit pattern for one line }
        for k := 0 to 7 do
          begin

```

```

        if ( BMask and 128 <> 0 ) then                { Pixel set? }
            setpix( x+k, y+i, fg );                    { Yes }
            BMask := BMask shl 1;
        end;
    end
else                { No --> consider background as well }
    for i := 0 to 7 do                { Execute lines }
        begin
            BMask := fptr^[ord(thechar),i];{ Get bit pattern for one line }
            for k := 0 to 7 do
                begin
                    if ( BMask and 128 <> 0 ) then        { Foreground? }
                        setpix( x+k, y+i, fg )            { Yes }
                    else
                        setpix( x+k, y+i, bk );            { No --> Background }
                        BMask := BMask shl 1;
                    end;
                end;
            end;
        end;
    end;
end;

```

```

{ *****
*   PrintString: Writes a string to the screen in graphics mode.   *
*   -----*
*   Input      :   x, y      = X- and Y-coordinates of upper left-corner *
*                 FG        = Foreground color                        *
*                 BK        = Background color                        *
*                 TSTR      = String to be displayed                  *
*   Info       :   The characters are designed around an 8x8 matrix, based *
*                 on the 8x8 ROM font.                                *
*   *****}

```

```

procedure PrintString( x, y : integer; fg, bk : byte; tstr : string );

```

```

var i : integer;                                { Loop counter }

begin
  for i := 1 to length( tstr ) do                { Execute string }
    begin
      PrintChar( tstr[i], x, y, fg, bk );          { and display it }
      inc( x, 8 );                                { Increment output position }
    end;
  end;

{ *****
*   Line: Draws a line based on the Bresenham algorithm.           *
**-----**
*   Input   : X1, Y1 = Starting coordinates  (0 - ...)           *
*             X2, Y2 = Ending coordinates                        *
*             LPCOL  = Color of line pixels                      *
*****}

procedure Line( x1, y1, x2, y2 : integer; lpcol : byte );

var d, dx, dy,
    aincr, bincr,
    xincr, yincr,
    x, y                : integer;

{-- Procedure for swapping two integer variables -----}

procedure SwapInt( var i1, i2: integer );

var dummy : integer;

```

```

begin
    dummy := i2;
    i2     := i1;
    i1     := dummy;
end;

{-- Main procedure -----}

begin
    if ( abs(x2-x1) < abs(y2-y1) ) then          { X- or Y-axis overflow? }
    begin                                         { Check Y-axes }
        if ( y1 > y2 ) then                      { y1 > y2? }
        begin
            SwapInt( x1, x2 );                  { Yes --> Swap X1 with X2 }
            SwapInt( y1, y2 );                  { and Y1 with Y2 }
        end;

        if ( x2 > x1 ) then xincr := 1          { Set X-axis increment }
        else xincr := -1;

        dy := y2 - y1;
        dx := abs( x2-x1 );
        d  := 2 * dx - dy;
        aincr := 2 * (dx - dy);
        bincr := 2 * dx;
        x := x1;
        y := y1;

        setpix( x, y, lpcol );                  { Set first pixel }
        for y:=y1+1 to y2 do                    { Execute line on Y-axes }
        begin
            if ( d >= 0 ) then

```



```

        begin
            inc( x, xincr );
            inc( d, aincr );
        end
    else
        inc( d, bincr );
        setpix( x, y, lpcol );
    end;
end
else
begin
    { Check X-axes }
    if ( x1 > x2 ) then
        { x1 > x2? }
        begin
            SwapInt( x1, x2 );
            { Yes --> Swap X1 with X2 }
            SwapInt( y1, y2 );
            { and Y1 with Y2 }
        end;

    if ( y2 > y1 ) then yincr := 1
        { Set Y-axis increment }
        else yincr := -1;

    dx := x2 - x1;
    dy := abs( y2-y1 );
    d := 2 * dy - dx;
    aincr := 2 * (dy - dx);
    bincr := 2 * dy;
    x := x1;
    y := y1;

    setpix( x, y, lpcol );
    { Set first pixel }
    for x:=x1+1 to x2 do
        { Execute line on X-axes }
        begin
            if ( d >= 0 ) then

```

```

        begin
            inc( y, yincr );
            inc( d, aincr );
        end
    else
        inc( d, bincr );
        setpix( x, y, lpcol );
    end;
end;
end;

```

```

{ ****
*   GetVideo: Gets the contents of a rectangular range from the video   *
*               RAM and puts them in a buffer                           *
* ****-----**
*   Input      : PAGE      = Screen page (0 or 1)                       *
*               X1, Y1     = Starting coordinates                       *
*               WRANGE     = Width of the rectangular range in pixels   *
*               HRANGE     = Height of rectangular range in pixels      *
*               BUFPTR     = Pointer to pixel buffer, in which the inform- *
*                           ation is to be placed                       *
*   Output     : Pointer to created pixel buffer with the contents of    *
*               the specified area                                       *
*   Info       : If the value NIL is passed for the BUFPTR parameter,   *
*               a new pixel buffer is allocated via the heap and        *
*               returned. This buffer can be specified again for a new  *
*               call, unless the previous contents are still required   *
*               and the size of the rectangular area remains unchanged  *
*               compared to the preceding call.                         *
*               The specified area must begin at an X-coordinate that   *
*               can be divided by eight and extend over a multiple of  *
*               eight pixels.                                           *
* ****

```

```

*****}

function GetVideo( page : byte; x1, y1 : integer;
                  wrange, hrange : byte; bufptr : PIXPTR ) : PIXPTR;

begin
  if ( bufptr = NIL ) then
    begin
      { No buffer passed during call? }
      { No, create one }
      new( bufptr );
      { Create pixel buffer }
      getmem( bufptr^.pixbptr, (wrange*hrange) div 2 ); { Alloc. px.b. }
      bufptr^.numrows := hrange;
      { Height of buffer in lines }
      bufptr^.widthbytes := wrange div 8;
      { Width of a line in bytes }
      bufptr^.pixblen := (wrange*hrange) div 2;
      {Total len. of buffer }
    end;

    CopyVideo2Buf( bufptr^.pixbptr, page, x1, y1, wrange div 8, hrange );
    GetVideo := bufptr;
    { Returns pointer and buffer to caller }
  end;

{ *****
* PutVideo: Writes the contents of a rectangular area of the screen *
* previously saved by GetVideo back to the video RAM *
*-----*
* Input : BUFPTR = Pointer to pixel buffer returned during *
* previous call for GetVideo *
* PAGE = Screen page (0 or 1) *
* X1, Y1 = Starting coordinates *
* Info : This procedure does not delete the pixel buffer. The *
* FreePixBuf procedure must be called for this. *
* The specified X-ordinate must be a multiple of eight! *
* *****}

```

```

procedure PutVideo( bufptr : PIXPTR; page : byte; x1, y1 : integer );

begin
  CopyBuf2Video( bufptr^.pixbptr, page, x1, y1,
    bufptr^.widthbytes, bufptr^.numrows );
end;

{ *****
*   FreePixBuf: Deletes a pixel buffer, which was allocated previously *
*               via the heap when GetVideo was called.                 *
*   ----- *
*   Input      : BUFPTR = Pointer to pixel buffer returned during      *
*               previous call for GetVideo                             *
*   ----- *
*   ----- *
*****}

procedure FreePixBuf( bufptr : PIXPTR );

begin
  freemem( bufptr^.pixbptr, bufptr^.pixblen );
  dispose( bufptr );
end;

{ *****
*   CreateSprite: Creates a sprite based on a user-defined             *
*               pixel pattern.                                          *
*   ----- *
*   Input       : SPLLOOKP = Pointer to data structure from CompileSprite *
*   Output      : Pointer to created sprite structure                  *
*   ----- *
*****}

function CreateSprite( splookp : SPLP ) : SPIP;

```

```

var spidp : SPIP;                                { Pointer to created sprite structure }

begin
  new( spidp );                                  { Allocate memory for sprite descriptor }
  spidp^.splookp := splookp;                     { Pass data to the sprite structure }
  {-- Create two background buffers by saving a large enough area
    { from the video RAM via GetVideo }

  spidp^.hgptr[0] := GetVideo( 0, 0, 0, splookp^.twidth,
                               splookp^.theight, NIL );
  spidp^.hgptr[1] := GetVideo( 0, 0, 0, splookp^.twidth,
                               splookp^.theight, NIL );

  CreateSprite := spidp;                         { Return pointer to the sprite structure }
end;

{*****
* CompileSprite: Creates a sprite's pixel and bit patterns, based on
*               the sprite's definition at runtime.
*-----*****
* Input      : BUFP      = Pointer to array contains string pointers
*               controlling sprite's pattern
*               SHEIGHT = Sprite height (and number of strings needed)
* Info       : In passed sprite pattern, a space stands for a back-
*               ground pixel, the A stands for the color code 0, B for
*               1, C for 2 etc.
*****}

function CompileSprite( var buf; sheight : byte ) : SPLP;

type BYPTR = ^byte;                                { Pointer to a byte }

```

```

var  stwidth,                                { String width }
     spwidth,                                { Sprite width }
     c,                                     { get character from c sprite array }
     i, k, l, y,                             { Loop variables }
     andc,                                  { Pixel counter for creating the bit mask }
     andm      : byte;                      { Pixel mask }
     andindex  : integer;                   { Index in AND buffer }
     splookp   : SPLP;                     { Pointer to created sprite structure }
     lspb      : BYPTR;                     { Floating pointer in sprite buffer }
     andp,     { Pointer to AND buffer }
     bptr      : BARPTR;                   { Addresses buffer with graphic }
     tpix      : PIXPTR;                   { Pointer to temporary pixel buffer }

```

```

{-- Sub-procedure AndBufInit: Initializes an AND buffer -----}

```

```

procedure AndBufInit( bufp : BARPTR );

```

```

begin
    andp := bufp;                { Note pointer to buffer }
    andindex := 0;               { Start at beginning of buffer }
    andm := 0;                   { First bit mask is 0 }
    andc := 0;                   { Still no bit in first byte }
end;

```

```

{-- Sub-procedure AndBufAppendBit: Add a bit to the AND buffer -----}

```

```

procedure AndBufAppendBit( bit : byte );

```

```

begin
    andm := andm or bit;         { Set bit in bit position 0 }
    if andc = 7 then             { Byte full now? }
        begin                    { Yes }

```

```

        andp^[andindex] := andm;           { Place byte in buffer }
        inc( andindex );                   { address next byte }
        andm := 0;                         { Bit mask back to 0 }
        andc := 0;                         { Continue in next byte with first bit }
    end
else                                       { Byte is not yet full }
    begin
        inc( andc );                       { One more bit processed }
        andm := andm shl 1;                { Shift bit mask }
    end;
end;

{-- Sub-procedure AndBufEnd: Close AND buffer -----}

procedure AndBufEnd;

begin
    if ( andc <> 0 ) then                   { Last byte not yet full? }
        andp^[andindex] := andm shl (7 - andc); { No, close }
    end;

begin
    {-- Create Sprite-Look structure and fill with data -----}

    new( splookp );
    bptr := @buf;                          { Set pointer to logo buffer }
    stwidth := bptr^[0];                    { Get string length and determine log width }
    spwidth := ( ( stwidth + 7 + 7 ) div 8 ) * 8; { Total width }
    splookp^.twidth := spwidth;             { Note width and height }
    splookp^.theight := sheight;

    setpage( 1 );                          { Draw sprites in page 1 }

```

```

showpage( 0 );                                { but display page 0 }
tpix := GetVideo( 1, 0, 0, spwidth, sheight, NIL ); { Note backgrnd }

{-- Draw and code sprite eight times -----}

for l := 0 to 7 do
  begin
    { First fill background with black pixels }
    for y := 0 to sheight-1 do
      Line( 0, y, spwidth-1, y, 0 );

    {-- Create and initialize memory for AND buffer -----}

    getmem( splookp^.bmskp[ 1 ], (spwidth*sheight) div 8 );
    AndBufInit( splookp^.bmskp[ 1 ] );

    for i := 0 to sheight-1 do
      begin
        { Execute lines }
        for y := 1 to l do
          { Create AND bits for left margins }
          AndBufAppendBit( 1 );

        for k := 0 to stwidth-1 do
          { Execute columns }
          begin
            c := bptr^[i*(stwidth+1)+k+1];
            { Get color }
            if ( c = 32 ) then
              { Background pixel? }
              begin
                { Yes --> set color code 0 }
                setpix( k+1, i, 0 );
                AndBufAppendBit( 1 );
                { Background pixel stays }
              end
            else
              { No, set color code as given }
              begin
                setpix( k+1, i, c-ord('@') );
                AndBufAppendBit( 0 );
                { Mask background pixel }
              end
            end
          end
        end
      end
    end
  end

```



```

        end;
    end;
    for y := spwidth-stwidth-1 downto 1 do    { Add AND bits for }
        AndBufAppendBit( 1 );                { the right margin }
    end;
    AndBufEnd;                                { Close AND buffer }

    {-- Get sprite's pixel pattern from video RAM -----}
    splookp^.pixmp[ 1 ] := GetVideo( 1, 0, 0, spwidth, sheight, nil );

    end;                                       { Draw next of eight sprites }

    PutVideo( tpix, 1, 0, 0 );                { Restore sprite background in }
    FreePixBuf( tpix );                      { page 1 and delete buffer }

    CompileSprite := splookp;                { Return pointer to sprite buffer }
end;

{ *****
*   PrintSprite : Displays sprite in a specified page   *
*-----*
*   Input      : SPIDP  = Pointer to the sprite structure
*                SPRPAGE = Page in which sprite should be drawn ( 0 or 1 )
* ***** }

procedure PrintSprite( spidp : SPIP; sprpage : byte );

var x : integer;                            { X-coordinate of sprite }

begin
    x := spidp^.x[sprpage];
    MergeAndCopyBuf2Video( spidp^.splookp^.pixmp[x mod 8]^ .pixbptr,

```

```

        spidp^.hgptr[sprpage]^..pixbptr,
        spidp^.splookp^.bmskp[x mod 8],
        sprpage,
        x and not(7),
        spidp^.y[sprpage],
        spidp^.splookp^.twidth div 8,
        spidp^.splookp^.theight );
end;

{ *****
*   GetSpriteBg: Gets a sprite background and specifies the position.   *
*   -----*
*   Input      : SPIDP      = Pointer to the sprite structure           *
*                SPRPAGE    = Page from which the background should be  *
*                (0 or 1)                                           *
*   -----*
*   *****}

procedure GetSpriteBg( spidp : SPIP; sprpage : BYTE );

var dummy : PIXPTR;

begin
    dummy := GetVideo( sprpage, spidp^.x[sprpage] and not(7), spidp^.y[sprpage],
        spidp^.splookp^.twidth, spidp^.splookp^.theight,
        spidp^.hgptr[sprpage] );
end;

{ *****
*   RestoreSpriteBg: Restores sprite background from original graphic   *
*                page.                                           *
*   -----*
*   Input      : SPIDP      = Pointer to the sprite structure           *
*                -----*
*   -----*
*   *****}

```

```

*          SPRPAGE = Page from which background should be copied      *
*          (0 or 1)                                                    *
*****}

procedure RestoreSpriteBg( spidp : SPIP; sprpage : BYTE );

begin
  PutVideo( spidp^.hgp[ptr[sprpage], sprpage,
            spidp^.x[sprpage] and not(7), spidp^.y[sprpage] );
end;

{*****}
* MoveSprite: Copy sprite within background to original graphic page.*
*-----*
* Input   : SPIDP = Pointer to the sprite structure                  *
*           SPRPAGE= Page to which the background should be copied  *
*           (0 or 1)                                                  *
*           DELTAX = Movement counter in X-                          *
*           DELTAY  and Y-directions                                *
* Output   : Collision marker (See OUT_ constants)                  *
*****}

function MoveSprite( spidp : SPIP; sprpage : byte;
                    deltax, deltax : integer      ) : byte;

var newx, newy : integer;                      { New sprite coordinates }
    out       : byte;                         { Display collision with border }

begin
  {-- Move X-coordinates and test for border collision -----}

  newx := spidp^.x[sprpage] + deltax;

```

```

if ( newx < 0 ) then
  begin
    newx := 0 - deltax - spidp^.x[sprpage];
    out := OUT_LEFT;
  end
else
  if ( newx > MAXX - spidp^.splookp^.twidth ) then
    begin
      newx := (2*(MAXX+1))-newx-2*(spidp^.splookp^.twidth);
      out := OUT_RIGHT;
    end
  else
    out := OUT_NO;

  {-- Move Y-coordinates and test for border collision -----}
  newy := spidp^.y[sprpage] + deltay;
  if ( newy < 0 ) then
    begin
      { Top border? }
      { Top border? }
      { Yes --> Deltay must be negative }
      newy := 0 - deltay - spidp^.y[sprpage];
      out := out or OUT_TOP;
    end
  else
    if ( newy + spidp^.splookp^.theight > MAXY+1 ) then
      begin
        { Bottom? }
        { Yes --> Deltay must be positive }
        newy := (2*(MAXY+1))-newy-2*(spidp^.splookp^.theight);
        out := out or OUT_BOTTOM;
      end;

  {-- Set new position only if different from old position -----}
  if ( newx <> spidp^.x[sprpage] ) or ( newy <> spidp^.y[sprpage] ) then

```

```

begin
    RestoreSpriteBg( spidp, sprpage );    { If there's a new position }
    spidp^.x[sprpage] := newx;            { then reset background and }
    spidp^.y[sprpage] := newy;            { store new coordinates      }
    GetSpriteBg( spidp, sprpage );        { Get new background      }
    PrintSprite( spidp, sprpage );        { Draw sprite in specified page }
end;

MoveSprite := out;
end;

{ *****
*   SetSprite: Sets sprite at a specific position.
*   -----
*   Input   : SPIDP = Pointer to the sprite structure
*             x0, y0 = Sprite coordinates for page 0
*             x1, y1 = Sprite coordinates for page 1
*   Info    : This function call should be made the first time that
*             MoveSprite() is called
*   ***** }

procedure SetSprite( spidp : SPIP; x0, y0, x1, y1 : integer );

begin
    spidp^.x[0] := x0;    { Store coordinates in sprite structure }
    spidp^.x[1] := x1;
    spidp^.y[0] := y0;
    spidp^.y[1] := y1;

    GetSpriteBg( spidp, 0 );    { Get sprite backgrounds }
    GetSpriteBg( spidp, 1 );    { in pages 0 and 1      }
    PrintSprite( spidp, 0 );    { Draw sprite in      }

```

```
PrintSprite( spidp, 1 ); { pages 1 and 0 }
end;
```

```
{*****
* Demo: Demonstrates these functions. *
*****}
```

```
procedure Demo;
```

```
const StarShipUp :array [1..20] of string[32] =
(
    '          AA          ',
    '          AAAA         ',
    '          AAAA         ',
    '          AA           ',
    '          GGBBGG        ',
    '          GBCCBBG       ',
    '          GBBBCCBBBG    ',
    '          GBEBBBBBBBBG   ',
    '          GBEBBBBBBBBG   ',
    ' G          GBEBBBBBBBBG   G ',
    'GCG          GGDDBBBBBBBBDGG   GCG',
    'GCG  GGBBDBBB  BBDBBBBGG  GCG',
    'GCBGGGBBBBBDBB  BBDBBBBBGGGBCG',
    'GCBBBBBBBBBBDB  BDBBBBBBBBBBCG',
    'BBBBBBBBBBBBBDB BB  BDBBBBBBBBBBBB',
    'GGBBBBBBBBDBBBBBBBBBBDBBBBBBBCG',
    '  GGCCBBDDDDDDDDDDDDDBBCCG ',
    '    GGBDDDDDDGGGGDDDDDBBG   ',
    '      GDDDDGGG    GGGDDDDG   ',
    '      DDDD        DDDD        ');
```

```
const StarShipDown :array [1..20] of string[32] =
```

```

( '      DDDD      DDDD      ' ,
  '      GDDDDGGG      GGGDDDDG      ' ,
  '      GGBBDDDDDDGGGGDDDDDBBG      ' ,
  '      GGCCBBBDDDDDDDDDDDBBBCCG      ' ,
  'GGBBBBBBBBBBBBBBBBBBBBBBBBBBCG      ' ,
  'BBBBBBBBBBBBBDB BB BDBBBBBBBBBBBB      ' ,
  'GCBBBBBBBBBBDB      BDBBBBBBBBBBCG      ' ,
  'GCBGGGBBBBBBDB      BDBBBBBGGGBCG      ' ,
  'GCG      GGBBDBBB      BBDBBBGG      GCG      ' ,
  'GCG      GGDBBBBBBBBBDGG      GCG      ' ,
  ' G      GBBBBBBBBBBBBG      G      ' ,
  '      GBBBBBBBBBBG      ' ,
  '      GBBBBBBBBBBG      ' ,
  '      GBBCCBBG      ' ,
  '      GBBCCBBG      ' ,
  '      GBBGG      ' ,
  '      AA      ' ,
  '      AAAA      ' ,
  '      AAAA      ' ,
  '      AA      ' );

```

```

SPRNUM = 6;                                { Number of sprites }
CWIDTH = 42;                              { Width of copyright message in characters }
CHEIGHT = 6;                              { Message height in rows }
SX      = (MAXX-(CWIDTH*8)) div 2;         { Starting X-coordinate }
SY      = (MAXY-(CHEIGHT*8)) div 2;        { Starting Y-coordinate }

```

```

type SPRITE = record                      { For sprite management }
    spidp : SPIP;                        { Pointer to sprite Id }
    deltax,                                { X-movement for pages 0 and 1 }
    deltay : array [0..1] of integer;    { Y-movement }
end;

```

```

var sprites      : array [1..SPRNUM] of SPRITE;
  page,
  lc,
  out              : byte;
  x, y, i,
  dx, dy          : integer;
  starshipupp,
  starshipdnp    : SPLP;
  ch              : char;
                                { Current page }
                                { Character for screen design }
                                { Get flags for page collision }
                                { Loop counter }
                                { Movement value }
                                { Sprite pointer }

begin
  Randomize;                    { Initialize random number generator }

  {-- Create patterns for the different sprites -----}

  starshipupp := CompileSprite( StarShipUp,  20 );
  starshipdnp := CompileSprite( StarShipDown, 20 );

  {-- Fill the first two graphic pages with characters -----}

  for page := 0 to 1 do
    begin
      setpage( page );
      showpage( page );
      lc := 0;
      y := 0;
      while ( y < (MAXY+1)-8 ) do
        begin
          x := 0;
          while ( x < (MAXX+1)-8 ) do
            begin

```



```

        PrintChar( chr(lc and 127), x, y, lc and 15, 0 );
        inc( lc );
        inc( x, 8 );
    end;
    inc( y, 12 );
end;

{-- Display copyright message -----}

Line( SX-1, SY-1, SX+CWIDTH*8, SY-1, 15 );
Line( SX+CWIDTH*8, SY-1, SX+CWIDTH*8, SY+HEIGHT*8, 15 );
Line( SX+CWIDTH*8, SY+HEIGHT*8, SX-1, SY+HEIGHT*8, 15 );
Line( SX-1, SY+HEIGHT*8, SX-1, SY-1, 15 );
PrintString( SX, SY, 15, 4,
            '
PrintString( SX, SY+8, 15, 4,
            ' S6435P.PAS - (c) 1992 by Michael Tischer ' );
PrintString( SX, SY+16, 15, 4,
            '
PrintString( SX, SY+24, 15, 4,
            ' Sprite demo for 640x350 mode ' );
PrintString( SX, SY+32, 15, 4,
            ' on EGA and VGA cards ' );
PrintString( SX, SY+40, 15, 4,
            '
end;

{-- Create different sprites -----}

for i := 1 to SPRNUM do
begin
    sprites[ i ].spidp := CreateSprite( starshipupp );

```

```

repeat                                { Select movement values for sprites }
    dx := 0;
    dy := random(10) - 5;
until ( dx <> 0 ) or ( dy <> 0 );

sprites[ i ].deltax[0] := dx * 2;
sprites[ i ].deltay[0] := dy * 2;
sprites[ i ].deltax[1] := dx * 2;
sprites[ i ].deltay[1] := dy * 2;

x := ( ((MAXX+1) div SPRNUM) * (i-1) )
    + (((MAXX+1) div SPRNUM)-40) div 2;
y := random( (MAXY+1) - 40 );
SetSprite( sprites[ i ].spidp, x, y, x - dx, y - dy );
end;

{-- Move sprites and bounce them off the page borders -----}

page := 1;                                { Start with page 1 }
while ( not keypressed ) do                { Press a key to end the loop }
begin
    showpage( 1-page );                    { Display other page }
    { ch := readkey; }                      { Remove key from keyboard buffer }
    for i := 1 to SPRNUM do                 { Execute sprites }
        begin                               { Move sprite and check for page collision }
            out := MoveSprite( sprites[i].spidp, page,
                               sprites[i].deltax[page],
                               sprites[i].deltay[page] );
            if ( ( out and OUT_TOP ) <> 0 ) or { Top/bottom collision? }
                ( ( out and OUT_BOTTOM ) <> 0 ) then
                begin
                    {-- Yes --> Change direction of movement -----}

```

```

        {----- and change sprite graphic -----}

        sprites[i].deltay[page] := -sprites[i].deltay[page];
        if ( ( out and OUT_TOP ) <> 0 ) then
            sprites[i].spidp^.splookp := starshipdnp
        else
            sprites[i].spidp^.splookp := starshippupp;
        end;
        if ( ( out and OUT_LEFT ) <> 0 ) or { Left/right collision? }
            ( ( out and OUT_RIGHT ) <> 0 ) then
            sprites[i].deltax[page] := -sprites[i].deltax[page];
        end;
        page := (page+1) and 1;                { Toggle between 1 and 0 }
    end;
    ch := readkey;                            { Remove key from keyboard buffer }
end;

{-----}
{--                M A I N    P R O G R A M                ----}
{-----}

begin
    if ( IsEgaVga <> NEITHERNOR ) then        { EGA or VGA card installed? }
        begin
            init640350;                        { Yes --> Go ahead }
            Demo;                             { Initialize graphic mode }
            Textmode( CO80 );                  { Shift into text mode }
        end
    else
        writeln( 'S6435P.PAS - (c) 1992 by Michael Tischer'#13#10#10 +
            'This program requires an EGA or a VGA card'#13#10 );
    end.

```