

```

{*****
*
*                               S 3 2 4 0 P . P A S
*
**-----**
* Task                        : Demonstrates sprites in 320x400 VGA graphic
*                            mode, using 256 colors and four screen pages.
*                            This program requires the assembly language
*                            modules V3240PA.ASM and S3240PA.ASM.
*
**-----**
* Author                      : Michael Tischer
* Developed on                 : 09/12/90
* Last update                  : 02/26/92
*****}

```

```
program S3240P;
```

```
uses dos, crt;
```

```
{-- External references from assembler routines -----}
```

```
{ $L v3240pa } { Add assembler module }
```

```

procedure init320400; external;
procedure setpix( x, y : integer; pcolor : byte ); external;
function getpix( x, y: integer ) : byte ; external;
procedure setpage( page : byte ); external;
procedure showpage( page : byte ); external;

```

```
{ $L s3240pa } { Add assembler module }
```

```

procedure CopyPlane2Buf( bufptr : pointer;
                        page      : byte;
                        fromx,

```

```

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

procedure CopyBuf2Plane( bufptr : pointer;
                        page   : byte;
                        tox,
                        toy     : integer;
                        rwidth,
                        rheight : byte;
                        bg      : boolean ); external;

{-- Constants -----}

const MAXX = 319;                { Maximum X- and Y-coordinates }
      MAXY = 399;

      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 }
    bitptr : array[0..3] of pointer; { Ptr. to bitplanes }
    byprod : array[0..3] of byte;    { Num. ranges to copy }
    rhght  : byte;                  { Number of rows }
    {-- Here follow bytes from the bitplanes -----}
end;
PIXPTR = ^PIXBUF;                { Pointer to a pixel buffer }

```

```

SPLOOK = record                                { Sprite design }
    twidth,                                    { Total width }
    theight : byte;                            { Height in pixel lines }
    pixbp   : PIXPTR;                          { Pointer to pixel block }
end;                                           { for allocating sprite }
SPLP = ^SPLOOK;                               { Pointer to sprite design }

```

```

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

```

```

PTRREC = record                                { For pointer or LONGINTS analysis }
    ofs,
    seg : word;
end;

```

```

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

```

```

{ *****
*   IsVga : Determines whether a VGA card is installed.   *
*-----*
*   Input   : None                                         *
*   Output  : TRUE or FALSE                               *
*-----*
}
```

```

function IsVga : boolean;

```

```

var Regs : Registers;           { Processor registers for interrupt call }

begin
  Regs.AX := $1a00;              { Function 1AH applies only to VGA }
  Intr( $10, Regs );
  IsVga := ( Regs.AL = $1a );
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 }
  Regs.AL := $30;                                { 11H, sub-function 30H }
  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;

{*****
*   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;
            end;
        end;
    end;
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;
        end;
    end;

```



```

if ( y2 > y1 ) then yincr := 1           { Set X-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;

{ *****
* GrfxPrint: Displays a formatted string on the graphic screen. *
*-----**
* Input   : X, Y   = Starting coordinates (0 - ...)           *
*          fg      = Foreground color                         *
*          bk      = Background color (255 = transparent)     *

```

```

*          STRING = String with format information          *
*****}

procedure GrfxPrint( x, y : integer; fg, bk : byte; strt : string );

var i : integer;                                     { Loop counter }

begin
  for i:=1 to length( strt ) do
    begin
      printchar( strt[i], x, y, fg, bk );    { Display using PrintChar }
      inc( x, 8 );                          { Move X to next character position }
    end;
  end;

{ *****
*   GetVideo: Places contents of a rectangular range of video RAM   *
*   in a buffer.                                                    *
**-----**
*   Input    : PAGE      = Screen page (0 or 1)                      *
*              X1, Y1    = Starting coordinates                      *
*              WRANGE    = Width of rectangular range in pixels      *
*              HRANGE    = Height of rectangular range in pixels     *
*              BUFPTR    = Pointer to pixel buffer into which        *
*                        information should be allocated              *
*   Output   : Pointer to allocated pixel buffer with contents of    *
*              specified range                                       *
*   Info     : If the BUFPTR parameter contains nothing, a new pixel *
*              buffer is allocated and returned on the heap. This buffer *
*              can be re-created on each call, provided the previous  *
*              contents are no longer needed, and provided the size   *
*              of the rectangular range remains unchanged.           *

```

```
*****}
```

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

```
var i,                                { Loop counter }  
    curplane,                        { Currently processed bitplane }  
    sb,                              { Bitplane at starting coordinates }  
    eb,                              { Bitplane at ending coordinates }  
    b,                              { Number of bytes per bitplane }  
    am : byte; { Number of bytes in center of both groups of four }  
    rptr : pointer;                 { Offset counter }
```

```
begin  
  if ( bufptr = NIL ) then           { No buffer passed on call? }  
    getmem( bufptr, sizeof( PIXBUF ) + wrange*hrange ); { No --> Alloc }
```

```
{-- Compute number of bytes per bitplane -----}
```

```
am := ( ( (x1+wrange-1) and not(3) ) - { Number of bytes in center }  
        ( (x1+4) and not(3) ) ) div 4;
```

```
sb := x1 mod 4;                      { Starting bitplane }  
eb := (x1+wrange-1) mod 4;          { Ending bitplane }
```

```
rptr := ptr( seg(bufptr^), ofs(bufptr^)+ sizeof( PIXBUF ) );
```

```
{-- Execute four bitplanes -----}
```

```
for i:=0 to 3 do
```

```
  begin  
    curplane := (sb+i) mod 4;  
    b := am; { Base number of bytes to be copied }
```

```

    if ( curplane >= sb ) then      { Also in starting block of four? }
        inc( b );                  { Yes --> Add a byte to this bitplane }
    if ( curplane <= eb ) then      { Also in ending block of four? }
        inc( b );                  { Yes --> Add a byte to this bitplane }
    bufptr^.bitptr[i] := rptr;      { Place pointer at start of buffer }
    bufptr^.byprod[i] := b;         { Place number in buffer }
    CopyPlane2Buf( rptr, page, x1+i, { Get contents
                                     y1, b, hrange ); { of bitplane }
    inc( PTRREC(rptr).ofs, b * hrange ); { Set pointer to next
end;                                { bitplane in buffer }

bufptr^.rhght := hrange;           { Store height }

GetVideo := bufptr;                { Return buffer pointer to caller }
end;

{ *****
* PutVideo: Writes contents of a rectangular screen range generated *
* by GetVideo to video RAM. *
*-----*
* Input   : BUFPTR = Pointer to pixel buffer from GetVideo *
*          PAGE    = Screen page (0 or 1) *
*          X1, Y1   = Starting coordinates *
*          BG       = Should background pixels (color code 255) not *
*                   be written to video RAM *
* Info    : Pixel buffer is not cleared by this procedure; use the *
*          FreePixBuf for this purpose. *
* ***** }

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

```

```

var curplane,                                { Currently executed bitplane }
    hrange : byte;

begin
    hrange := bufptr^.rhght;                  { Range height }
    for curplane:=0 to 3 do                    { Execute four bitplanes }
        CopyBuf2Plane( bufptr^.bitptr[curplane], page, xl+curplane,
                        y1, bufptr^.byprod[curplane], hrange, bg );
    end;

{ *****
*   FreePixBuf: Clears a pixel buffer previously allocated by Heap.   *
*   -----*
*   Input    : BUFPTR = Pointer to pixel buffer                      *
*               WRANGE = Width of rectangular range of pixels        *
*               HRANGE = Height of rectangular range of pixels       *
*   -----*
*   *****}

procedure FreePixBuf( bufptr : PIXPTR; wrange, hrange : byte );

begin
    freemem( bufptr, sizeof( PIXBUF ) + wrange*hrange );
end;

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

```



[illegible]

```

        setpix( k, i, fgcolor+(c-ord(fb)) );
    end;

    {-- Get sprite in buffer and restore background -----}

    splookp^.pixbp := GetVideo( gpage, 0, 0, swidth, sheight, NIL );
    PutVideo( pbptr, gpage, 0, 0, false );
    FreePixBuf( pbptr, swidth, sheight );          { Free 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 );

begin
    PutVideo( spidp^.splookp^.pixbp,
              sprpage, spidp^.x[sprpage], spidp^.y[sprpage], true );
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 taken *
*              (0 or 1)                                     *
*****}

```



```
*****}
```

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

```
var dummy : PIXPTR;
```

```
begin
```

```
  dummy := GetVideo( sprpage, spidp^.x[sprpage], spidp^.y[sprpage],  
                    spidp^.splookp^.twidth, spidp^.splookp^.theight,  
                    spidp^.bgptr[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^.bgptr[sprpage], sprpage,  
            spidp^.x[sprpage], spidp^.y[sprpage], false );
```

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

```

```

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

```

```

procedure Demo;

```

```

const StarShipUp :array [1..20] of string[32] =
  ( '           AA           ',
    '           AAAA          ',
    '           AAAA          ',
    '           AA            ',
    '           GGBBGG         ' );

```

```

'          GBBCCBBG          ' ,
'          GBBCCBBG          ' ,
'          GBBBBBBBBBG      ' ,
'          GBBBBBBBBBG      ' ,
' G          GBBBBBBBBBBG      G ' ,
'GCG      GGBBBDBBB      BBBDBBBGG      GCG' ,
'GCBGGGBBBBBDBB      BBDBBBBGGGBCG' ,
'GCBBBBBBBBBDB      BDBBBBBBBBBBCG' ,
'BBBBBBBBBBBBBDB BB      BDBBBBBBBBBBBB' ,
'GGBBBBBBBDBBBBBBBBBBDBBBBBBBBCG' ,
'      GGCCBBBDDDDDDDDDDDDDBBCCG' ,
'      GGBDDDDGGGGGDDDDDBBG      ' ,
'      GDDDDGGG      GGGDDDDG      ' ,
'          DDDD          DDDD      ' );

```

```

const StarShipDown :array [1..20] of string[32] =
(
'          DDDD          DDDD      ' ,
'      GDDDDGGG      GGGDDDDG      ' ,
'      GGBDDDDGGGGGDDDDDBBG      ' ,
'      GGCCBBBDDDDDDDDDDDDDBBCCG      ' ,
'GGBBBBBBBDBBBBBBBBBBDBBBBBBBBCG' ,
'BBBBBBBBBBBBBDB BB      BDBBBBBBBBBBBB' ,
'GCBBBBBBBBBDB      BDBBBBBBBBBBCG' ,
'GCBGGGBBBBBDBB      BBDBBBBGGGBCG' ,
'GCG      GGBBBDBBB      BBBDBBBGG      GCG' ,
'GCG      GGBBBBBBBBBBBDDGG      GCG' ,
' G          GBBBBBBBBBBBG      G ' ,
'          GBBBBBBBBBBG      ' ,
'          GBBBBBBBBBBG      ' ,
'          GBBCCBBG      ' ,
'          GBBCCBBG      ' ,

```

```

      '          GBBGG          ' ,
      '          AA            ' ,
      '          AAAA          ' ,
      '          AAAA          ' ,
      '          AA            ' );

```

```

SPRNUM = 6;                                { Number of sprites }
CWIDTH = 37;                              { Width of copyright message in characters }
HEIGHT = 6;                                { Message height in rows }
SX      = (MAXX-(CWIDTH*8)) div 2;         { Starting X-coordinate }
SY      = (MAXY-(HEIGHT*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 }
    deltax : array [0..1] of integer;      { Y-movement }
    delay : array [0..1] of integer;
end;

var sprites      : array [1..sprnum] of SPRITE;
    page,
    lc,
    out           : byte;
    x, y, i,
    dx, dy        : integer;
    starshipupp,
    starshipdnp   : SPLP;
    ch             : char;
begin

    Randomize;                                { Initialize random number generator }

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

```

```

starshipupp := CompileSprite( StarShipUp, 20, 0, 'A', 1 );
starshipdnp := CompileSprite( StarShipDown, 20, 0, 'A', 1 );

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

for page := 0 to 1 do
begin
  setpage( 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 mod 255, 0 );
          inc( lc );
          inc( x, 8 );
        end;
        inc( y, 12 );
      end;
    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+CHEIGHT*8, 15 );
  Line( SX+CWIDTH*8, SY+CHEIGHT*8, SX-1, SY+CHEIGHT*8, 15 );
  Line( SX-1, SY+CHEIGHT*8, SX-1, SY-1, 15 );
  PrintString( SX, SY, 15, 4,
    '
    ');
  PrintString( SX, SY+8, 15, 4,

```

```

        ' * S3240P.PAS - (c) 1992 M. Tischer* ' );
PrintString( SX, SY+16, 15, 4,
        ' );
PrintString( SX, SY+24, 15, 4,
        ' Sprite demo for 320x400 mode ' );
PrintString( SX, SY+32, 15, 4,
        ' on 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 := ( (320 div SPRNUM) * (i-1) ) + ((320 div SPRNUM)-40) div 2;
    y := random( 200 - 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 }

    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 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 := starshipupp;
        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;                          { Wait for a key }
end;

```

```
{*****
*                               M A I N   P R O G R A M                               *
*****}
```

```
begin
  if ( IsVga ) then                { VGA card installed? }
    begin                          { Yes --> Go ahead }
      init320400;                  { Initialize graphic mode }
      Demo;
      Textmode( CO80 );            { Shift into text mode }
    end
  else
    writeln( 'S3240P.PAS - (c) 1992 by Michael Tischer'#13#10#10 +
      'This program requires a VGA card'#13#10 );
end.
```