

```

{*****
*   M I K A D O P : Demonstrates 512 character mode on EGA & VGA color *
*                   systems: Displays graphics within text mode.      *
*                   This program runs in VGA color mode only. If you  *
*                   are running a VGA mono system, switch your card to *
*                   VGA color mode before running this program.      *
*-----*
*   Author          : Michael Tischer                                *
*   Developed on    : 04/02/90                                         *
*   Last update     : 02/12/92                                         *
*****}

```

program MikadoP;

uses DOS, CRT; { Add DOS and CRT units }

{-- Constants -----}

```

const EGAVGA_SEQUENCER = $3C4; { Sequencer address/data port }
      EGAVGA_MONCTR    = $3D4; { Monitor controller }
      EGAVGA_GRAPHCTR  = $3CE; { Graphics controller address/data port }
      CHAR_WIDTH       = 8;
      CHAR_BYTES       = 32;
      MIKADOS          = 5; { Number of mikados drawn simultaneously }

```

{-- Type declarations -----}

```

type VEL = record { Describes character attribute }
               case boolean of { combination in video RAM }
                 true : ( Charctr, ChAttrib : byte );
                 false : ( Contnt          : word );
               end;

```

```

VPTR      = ^VEL;                { Pointer to a character/attribute }
VELARRAY  = array [1..25,1..80] of VEL;    { Video RAM array }
VELARPTR  = ^VELARRAY;           { Pointer to video RAM }

FONT = array[0..255,0..CHAR_BYTES-1] of byte; { Font array }
FPTR = ^font;                     { Pointer to a font }

PALARY = array[ 1..16] of BYTE;      { Palette register array }

{-- Global variables -----}

const vioptr : VELARPTR = ptr( $B800, $0000 ); { Pointer to video RAM }

var CharHeight,
    lenx      : byte;             { Width of graphic window in characters }
    xmax,     { Max. pixel coordinates of graphic window }
    ymax      : integer;
    fontptr   : fptr;             { Pointer to graphic font }

procedure CLI; inline( $FA );
procedure STI; inline( $FB );

{ *****
*   IsEgaVga : Determines whether an EGA or a VGA card is installed,
*               then places the number of scan lines per character in
*               the CharHeight global variable.
* *****
**-----**
*   Input    : None
*   Output    : TRUE if EGA or VGA card, otherwise FALSE
* *****
}

function IsEgaVga : boolean;

```

```

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

begin
  Regs.AX := $1a00;              { Function 1AH applies to VGA only }
  Intr( $10, Regs );
  if ( Regs.AL = $1a ) then      { Is the function available? }
    begin
      IsEgaVga := TRUE;
      CharHeight := 16;          { 16 scan lines }
    end
  else
    begin
      Regs.ah := $12;             { Call function 12H, }
      Regs.bl := $10;             { sub-function 10H }
      intr($10, Regs);            { Call video BIOS }
      IsEgaVga := ( Regs.bl <> $10 );
      CharHeight := 14;           { 14 scan lines }
    end;
end;

{ *****
*   SetCharWidth: Sets VGA character width to 8 or 9 pixels.   *
*   -----*
*   Input      : HWIDTH = Character width (8 or 9)             *
*   *****}

procedure SetCharWidth( hwidth : byte );

var Regs : Registers;           { Processor registers for interrupt call }
    x      : byte;              { Value for misc. output reg. }

```

```

begin
  if ( hwidth = 8 ) then Regs.BX := $0001      { BH = horiz. direction }
    else Regs.BX := $0800;      { BL = seq. reg. value }

  x := port[ $3CC ] and not(4+8);              { Toggle horizontal }
  if ( hwidth = 9 ) then                      { resolution from }
    x := x or 4;                              { 720 to 640 pixels }
  port[ $3C2 ] := x;

  CLI;                                         { Toggle sequencer from 8 to 9 pixels }
  portw[ EGAVGA_SEQUENCER ] := $0100;
  portw[ EGAVGA_SEQUENCER ] := $01 + Regs.BL shl 8;
  portw[ EGAVGA_SEQUENCER ] := $0300;
  STI;

  Regs.AX := $1000;                          { Change screen configuration }
  Regs.BL := $13;
  intr( $10, Regs );
end;

{ *****
*   SelectMaps : Selects fonts, with the selection depending on bit 3
*               of the attribute byte.
* *****
**-----**
*   Input      : MAP0 = Number of first font   ( bit 3 = 0 )
*               MAP1 = Number of second font   ( bit 3 = 1 )
*   Info       : EGA cards can select fonts 0-3,
*               VGA cards can select fonts 0-7.
* *****
}

procedure SelectMaps( map0, map1 : byte );

```

```

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

begin
  Regs.AX := $1103;              { Program font map select register }
  Regs.BL := ( ( map0 and 3 ) + ( map0 and 4 ) shl 2 ) +
    ( ( map1 and 3 ) shl 2 + ( map1 and 4 ) shl 3 );
  Intr( $10, Regs );            { Call BIOS function 11H, sub-function 03H }
end;

{ *****
*   GetFontAccess: Enables direct access to the second memory map in   *
*   which the font is stored at address A000:0000.                     *
*   -----*
*   Input   : None                                                    *
*   Info    : After calling this procedure you cannot access video RAM *
*               at B800:0000.                                          *
*   *****}

procedure GetFontAccess;

const SeqRegs : array[1..4] of word = ( $0100, $0402, $0704, $0300 );
      GCRegs   : array[1..3] of word = ( $0204, $0005, $0406 );

var i : byte;                  { Loop counter }

begin
  CLI;
  for i := 1 to 4 do           { Load different sequencer registers }
    portw[ EGA_VGA_SEQUENCER ] := SeqRegs[ i ];

  for i := 1 to 3 do           { Load graphics controller registers }
    portw[ EGA_VGA_GRAPHCTR ] := GCRegs[ i ];

```

```
STI;
```

```
end;
```

```
{*****
*   ReleaseFontAccess: Releases access to video RAM at B800:0000, but   *
*   fonts in memory page #2 remain blocked.                             *
*-----**
*   Input   : None                                                       *
*****}
```

```
procedure ReleaseFontAccess;
```

```
const SeqRegs : array[1..4] of word = ( $0100, $0302, $0304, $0300 );
      GCRegs   : array[1..3] of word = ( $0004, $1005, $0E06 );
```

```
var i : byte;                                { Loop counter }
```

```
begin
```

```
  for i := 1 to 4 do                      { Load different sequencer registers }
    portw[ EGAVGA_SEQUENCER ] := SeqRegs[ i ];
  for i := 1 to 3 do                      { Load graphics controller registers }
    portw[ EGAVGA_GRAPHCTR ] := GCRegs[ i ];
```

```
end;
```

```
{*****
*   ClearGraphArea: Clears the graphic area in which the character     *
*   patterns of stored characters are set to 0.                         *
*-----**
*   Input   : None                                                       *
*****}
```

```
procedure ClearGraphArea;
```

```

var exchars,                               { Characters to be executed }
    chrow    : byte;                       { Row within the corresponding character }

begin
  for exchars := 0 to 255 do                { Loop characters }
    for chrow := 0 to CharHeight-1 do      { Loop rows }
      fontptr^[ exchars, chrow ] := 0;    { & set to 0 }
    end;
end;

{ *****
* InitGraphArea: Initializes a screen area for graphic display. *
*-----*
* Input   : X       = Starting column of area (1-80)             *
*          Y       = Starting row of area (1-25)                 *
*          XLEN    = Area width in characters                    *
*          YLEN    = Area height in characters                    *
*          MAP     = Number of the graphic font                  *
*          GACOL   = Graphic area color (0-7 or FFH)             *
* Info    : If a color value of FFH exists, the system generates an *
*          appropriate color code needed for the mikado effect.  *
*-----*
* ***** }

procedure InitGraphArea( x, y, xlen, ylen, map, gacol : byte );

var column, chrow : integer;                { Loop variables }
    ccode         : byte;                   { Floating character code }

begin
  if ( xlen * ylen > 256 ) then              { Range too large? }
    writeln( 'Error: Area larger than the 256-character maximum' )
  else

```

```

begin
  if ( CharHeight = 16 ) then                                { VGA? }
    SetCharWidth( 8 );                                       { Yes --> Set character width }
    SelectMaps( 0, map );                                    { Select font }
    xmax := xlen*CHAR_WIDTH;                                { Compute max. pixel coordinates }
    ymax := ylen*CharHeight;
    lenx := xlen;
    fontptr := ptr( $A000, map * $4000 ); { Pointer to graphic map }
    GetFontAccess;                                           { Enable font access }
    ClearGraphArea;                                           { Clear font }
    ReleaseFontAccess;                                       { Enable video RAM access }

    {-- Fill graphic area with character codes -----}

    ccode := 0;
    for chrow := ylen-1 downto 0 do                          { Rows from bottom to top }
      for column := 0 to xlen-1 do                          { Columns from left to right }
        begin                                                { Set character code and attribute }
          vioptr^[chrow+y,column+x].Chrattr := ccode;
          if ( gacol = $ff ) then
            vioptr^[chrow+y,column+x].ChAttrib := ccode mod 6 + 1 + 8
          else
            vioptr^[chrow+y,column+x].ChAttrib := gacol or $08;
          inc( ccode );                                       { Next character code }
        end;
      end;
    end;
end;

{*****
* CloseGraphArea: Closes graphic area.
* ****}
* Input : None
*
```



```

*****}

procedure CloseGraphArea;

begin
  ReleaseFontAccess;           { Release access to video RAM }
  SelectMaps( 0, 0 );          { Always display font 0 }
  if ( CharHeight = 16 ) then  { VGA? }
    SetCharWidth( 9 );         { Yes --> Set character width }
end;

{ *****
* SetPixel: Sets or unsets a pixel in the graphic window. *
*-----*
* Input      : X,Y      = Pixel coordinates (0-...) *
*              ON        = TRUE to set, FALSE to unset *
*-----*
*****}

procedure SetPixel( x, y : integer; on : boolean );

var charnum,           { Code for character at coordinates }
    line      : byte;   { Pixel line in the character }

begin
  if ( x < xmax ) and ( y < ymax ) then { Coordinates O.K.? }
    begin { Yes --> Compute character no. and line }
      charnum := ((x div CHAR_WIDTH) + (y div CharHeight * lenx));
      line     := CharHeight - ( y mod CharHeight ) - 1;
      if on then { Set or unset character? }
        fontptr^[charnum, line] := fontptr^[charnum, line] or
          1 shl (CHAR_WIDTH - 1 - ( x mod CHAR_WIDTH ) )
      else

```

```

        fontptr^[charnum, line] := fontptr^[charnum, line] and
            not( 1 shl (CHAR_WIDTH - 1 - ( x mod CHAR_WIDTH ) ) );
    end;
end;

{*****
*   Line: Draws a line within the graphic window, using the Bresenham   *
*   algorithm.                                                            *
*-----*
*   Input   : X1, Y1 = Starting coordinates (0 - ...)                  *
*            X2, Y2 = Ending coordinates                                *
*            ON    = TRUE to set pixel, FALSE to unset pixel           *
*****}

procedure Line( x1, y1, x2, y2 : integer; on : boolean );

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;

      SetPixel( x, y, on );               { 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 );
        SetPixel( x, y, on );
    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
                else yincr := -1;
            { Set Y-axis increment }

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

            SetPixel( x, y, on );
            { 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 );
        SetPixel( x, y, on );
    end;
end;
end;

{ *****
*   SetPalCol: Defines a color from the 16-part color palette or the
*               screen border (overscan) color.
*   -----
*   Input      : RegNr = Palette register number (0-15) or 16 for the
*               overscan color
*               Col    = Color value from 0 to 15
*   *****}

procedure SetPalCol( RegNr : byte; Col : byte );

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

begin
    Regs.AX := $1000;            { Video function 10H, sub-function 00H }
    Regs.BH := Col;              { Color value }
    Regs.BL := RegNr;            { Register number of attribute controller }
    intr( $10, Regs );           { Call BIOS video interrupt }
end;

{ *****
*   SetPalAry: Installs a new 16-color palette without changing the
*               screen border color.
*   -----
*   Input      : NewCol = Palette array of type PALARY
*   *****}

```

```

procedure SetPalAry( NewCol : PALARY );

var i : byte;                                { Loop counter }

begin
  for i := 1 to 16 do                        { Execute 16 entries in array }
    SetPalCol( i-1, NewCol[ i ] );          { Set corresponding colors }
  end;

{*****}
*  GetPalCol: Gets the contents of a palette register.                                *
**-----**
*  Input   : RegNr = Palette register number (0-15) or 16 for the                    *
*            overscan color                                                            *
*  Output  : Color value                                                                *
*  Info    : Alternate included for EGA cards, which do not support                  *
*            interrupt 10H, function 10H, sub-function 07H.                          *
{*****}

function GetPalCol( RegNr : byte ) : byte;

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

begin
  if ( CharHeight = 14 ) then                { EGA card? }
    GetPalCol := RegNr                       { Yes --> Cannot read palette registers }
  else                                         { No --> VGA }
    begin
      Regs.AX := $1007;                      { Video function 10H, sub-function 07H }
      Regs.BL := RegNr;                     { Register number of attribute controller }
      intr( $10, Regs );                    { Call BIOS video interrupt }
    end
  end;
end;

```

```

        GetPalCol := Regs.BH;          { Palette register contents are here }
    end;
end;

{ *****
*   GetPalArray: Gets contents of 16-color palette registers and places   *
*   these contents in an array for the caller.                             *
*   -----*
*   Input    : ColArray = Palette array of type PALARY, into which colors *
*               are placed                                                *
*   *****}

procedure GetPalArray( var ColArray : PALARY );

var i : byte;                                { Loop counter }

begin
    for i := 1 to 16 do                    { Execute 16 entries in array }
        ColArray[ i ] := GetPalCol( i-1 ); { Set corresponding colors }
    end;

{ *****
*   Mikado: Applies the procedures and functions in this program.         *
*   -----*
*   Input    : None                                                         *
*   *****}

procedure Mikado;

type lcoor = record                        { Get coordinates of a line }
    x1, y1,
    x2, y2 : integer;

```

```

        end;

const NewCols : PALARY =
    ( {----- Normal text character colors -----}
      BLACK,      { Formerly...   black }
      BLUE,       {               blue }
      GREEN,      {               green }
      RED,        {               cyan }
      CYAN,       {               red }
      MAGENTA,    {               magenta }
      YELLOW,     {               brown }
      WHITE,      {               light gray }
      {----- Graphic colors -----}
      LIGHTBLUE,  { Formerly dark gray }
      LIGHTGREEN, {               light blue }
      LIGHTRED,   {               light green }
      LIGHTCYAN,  {               light cyan }
      LIGHTMAGENTA, {               light red }
      BLUE,       {               light magenta }
      YELLOW,     {               yellow }
      WHITE );    {               white }

var i,                { Loop counter }
    first,            { Array index of most recent mikado }
    last  : integer;  { Array index of oldest mikado }
    clear : boolean;  { Clear mikados }
    lar   : array[1..MIKADOS] of lcoor; { Mikado array }
    OldCols: PALARY;  { Get old colors }

begin
    GetPalAry( OldCols );      { Get old colors }
    SetPalAry( NewCols );     { Install new color palette }

```



```

TextColor( 7 );
TextBackGround( 1 );
ClrScr;                                     { Clear screen }
GotoXY(1,1);                               { Fill with characters }
for i:=1 to 25*80-1 do                     { from default font }
  write( chr(32 + i mod 224) );

{-- Initialize graphic area and generate mikados -----}

GotoXY(27,6);
TextColor( 7 );
TextBackGround( 3 );
write('          M I K A D O          ');
GotoXY(27,6);
InitGraphArea( 27, 7, 25, 10, 1, $FF );
GetFontAccess;                             { Get access to font }

clear := false;                            { No mikados cleared yet }
first := 1;                                { Start with first array position }
last := 1;
repeat                                     { Mikado loop }
  if first = MIKADOS+1 then first := 1;    { Wraparound? }
  lar[first].x1 := random( xmax-1 );        { Create mikado }
  lar[first].x2 := random( xmax-1 );
  lar[first].y1 := random( ymax-1 );
  lar[first].y2 := random( ymax-1 );
  line( lar[first].x1, lar[first].y1,
        lar[first].x2, lar[first].y2, true ); { and draw it }
  inc( first );                             { Next Mikado }
  if first = MIKADOS+1 then clear := true;  { Already clear? }
  if clear then                             { Clear now? }
    begin                                   { Yes }

```

```

        line( lar[last].x1, lar[last].y1,
              lar[last].x2, lar[last].y2, false );
        inc( last );                                { Clear next Mikado }
        if last = MIKADOS+1 then last := 1;
    end;
until keypressed;                                { Repeat until user presses a key }

{-- End program -----}

CloseGraphArea;
SetPalAry( OldCols );                            { Restore old color palette }
GotoXY(1, 25 );
TextColor( 7 );
TextBackGround( 0 );
ClrEol;
writeln( 'System has reverted to old font.' );
end;

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

begin
    if IsEgaVga then                                { Is there an EGA or a VGA card installed? }
        Mikado                                        { Yes --> Execute demo }
    else                                              { No --> Program cannot be started }
        writeln( 'Warning: No EGA or VGA card found' );
    end.

```