

```

/*****
*
*                               M I K A D O C . C
*
**-----**
*   Task                : 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
*****/

```

```

/*-- Define constants and add include files -----*/

```

```

#include <dos.h>                                /* Add include files */
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

```

```

#ifdef __TURBOC__                                /* Compiling with Turbo C? */
    #define CLI()          disable()
    #define STI()          enable()
    #define outpw( p, w )  output( p, w )
    #ifndef inp
        #define outp( p, b )  outputb( p, b )
        #define inp( p )      inputb( p )
    #endif
#else                                              /* No --> Then QuickC 2.0 or MSC */
    #include <conio.h>

```

```

#define random(x)      (rand() % ( x + 1 ))
#define MK_FP(seg,ofs) ((void far *)\
                        (((unsigned long)(seg) << 16) | (ofs)))

#define CLI()          _disable()
#define STI()          _enable()
#endif

#define EGAVGA_SEQUENCER 0x3C4          /* Sequencer address/data port */
#define EGAVGA_MONCTR   0x3D4          /* Monitor controller */
#define EGAVGA_GRAPHCTR 0x3CE /* Graphics controller addr./data port*/

#define CHAR_WIDTH      8
#define CHAR_BYTES      32
#define MIKADOS         5 /* Number of mikados drawn simultaneously */

#define TRUE ( 0 == 0 )
#define FALSE ( 0 == 1 )

#define BLACK           0x00           /* Color attributes */
#define BLUE            0x01
#define GREEN           0x02
#define CYAN            0x03
#define RED             0x04
#define MAGENTA         0x05
#define BROWN          0x06
#define LIGHTGRAY      0x07
#define GRAY            0x01
#define LIGHTBLUE       0x09
#define LIGHTGREEN      0x0A
#define LIGHTCYAN       0x0B
#define LIGHTRED        0x0C
#define LIGHTMAGENTA    0x0D

```

```

#define YELLOW          0x0E
#define WHITE          0x0F

/*-- Type declarations -----*/
typedef unsigned char BYTE;
typedef BYTE BOOL;

typedef BYTE PALARY[16];          /* Palette register array */

/*-- Global variables -----*/
BYTE far *vioptr = (BYTE far *) 0xb8000000, /* Pointer to video RAM */
far *fontptr;          /* Pointer to graphic font */

BYTE CharHeight,
lenx;          /* Width of graphic window in characters */
int xmax,      /* Max. pixel coordinates of graphic window */
ymax;

/*****
* IsEgaVga : Determines whether an EGA or VGA card is installed.
*-----*
* Input   : None
* Output  : EGA, VGA or NEITHER
*****/

BYTE IsEgaVga( void )
{
    union REGS Regs;          /* Processor registers for interrupt call */

    Regs.x.ax = 0x1a00;          /* Function 1AH applies to VGA only */

```

```

int86( 0x10, &Regs, &Regs );
if ( Regs.h.al == 0x1a )          /* Is the function available? */
{                                  /* Yes --> It's VGA */
    CharHeight = 16;              /* VGA character height */
    return 1;
}
else
{
    CharHeight = 14;              /* EGA character height */
    Regs.h.ah = 0x12;             /* Call function 12H, */
    Regs.h.bl = 0x10;             /* sub-function 10H */
    int86(0x10, &Regs, &Regs );   /* Call video BIOS */
    return Regs.h.bl != 0x10;
}
}

```

```

/*****
*   SetCursor : Specifies screen cursor position.
*-----**
*   Input parameters : CURCOL = New cursor column (0-79)
*                     CUROW  = New cursor row (0-24)
*   Return value      : None
*****/

```

```

void SetCursor( BYTE curcol, BYTE curow )
{
    union REGS regs;              /* Processor registers for interrupt call */

    regs.h.ah = 2;                /* Function number: Set cursor */
    regs.h.bh = 0;                /* Access screen page 0 */
    regs.h.dh = curow;            /* Set row */
    regs.h.dl = curcol;           /* Set column */
}

```

```

    int86(0x10, &regs, &regs);                /* Call BIOS video interrupt */
}

/*****
*   PrintfAt : Displays a formatted string anywhere on the screen.
**-----**
*   Input   : COLUMN = Column
*             SCROW  = Row
*             CHCOL   = Character attribute
*             STRING  = Pointer to string
*   Output  : None
*   Info    : This function should only be called if the system running
*             this program contains an EGA card or a VGA card.
*****/

void PrintfAt( BYTE column, BYTE scrow, BYTE chcol, char * string, ... )
{
    va_list parameter;                        /* Parameter list for VA_... macros */
    char outbufr[255],                       /* Buffer for formatted string */
        *outptr;
    BYTE far *vptr;                          /* Pointer to video RAM */

    va_start( parameter, string );           /* Convert parameters */
    vsprintf( outbufr, string, parameter );  /* Format */

    vptr = (BYTE far *) MK_FP( 0xB800, column*2+scrow*160 );

    for ( outptr = outbufr; *outptr ; )      /* Execute string */
    {
        *vptr++ = *(outptr++);              /* Write character to video RAM */
        *vptr++ = chcol;                   /* Write attribute to video RAM */
    }
}

```

```

}

/*****
*  ClrScr:  Clears the screen.                                     *
**-----**
*  Input parameters : CHATT  = Character attribute                 *
*  Return value      : None                                       *
*****/

```

```

void ClrScr( BYTE chatt )
{
    BYTE far *vptr;                                     /* Pointer to video RAM */
    int  count = 2000;                                  /* Number of characters to be cleared */

    vptr = (BYTE far *) MK_FP( 0xB800, 0 ); /* Set pointer to video RAM */

    for ( ; count-->0; )                                /* Execute video RAM */
    {
        *vptr++ = ' ';                                  /* Write character to video RAM */
        *vptr++ = chatt;                                /* Write attribute to video RAM */
    }
}

```

```

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

```

```

void SetCharWidth( BYTE hwidth )
{
    union REGS Regs;                                     /* Processor registers for interrupt call */

```

```

unsigned char x;                                /* Value for misc. output reg. */

Regs.x.bx = ( hwidth == 8 ) ? 0x0001 : 0x0800;

x = inp( 0x3CC ) & (255-12);                    /* Toggle horizontal */
if ( hwidth == 9 )                               /* resolution from */
    x |= 4;                                       /* 720 to 640 pixels */
outpw( 0x3C2, x);

CLI();                                           /* Toggle sequencer from 8 to 9 pixels */
outpw( EGAVGA_SEQUENCER, 0x0100 );
outpw( EGAVGA_SEQUENCER, 0x01 + ( Regs.h.bl << 8 ) );
outpw( EGAVGA_SEQUENCER, 0x0300 );
STI();

Regs.x.ax = 0x1000;                             /* Change horizontal screen position */
Regs.h.bl = 0x13;
int86( 0x10, &Regs, &Regs );
}

/*****
*   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.
*****/

void SelectMaps( BYTE map0, BYTE map1)
{

```

```

union REGS Regs; /* Processor registers for interrupt call */

Regs.x.ax = 0x1103; /* Program font map select register */
Regs.h.bl = ( map0 & 3 ) + ( ( map0 & 4 ) << 2 ) +
            ( ( map1 & 3 ) << 2 ) + ( ( map1 & 4 ) << 3 );
int86( 0x10, &Regs, &Regs ); /* Call function 11H, sub-function 03H */
}

/*****
* 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.
*****/

void GetFontAccess( void )
{
    static unsigned SeqRegs[4] = { 0x0100, 0x0402, 0x0704, 0x0300 },
                    GCRegs[3]  = { 0x0204, 0x0005, 0x0006 };
    BYTE i; /* Loop counter */

    CLI(); /* Disable interrupts */

    for ( i=0; i<4; ++i ) /* Load different sequencer registers */
        outpw( EGAVGA_SEQUENCER, SeqRegs[ i ] );

    for ( i=0; i<3; ++i ) /* Load graphics controller registers */
        outpw( EGAVGA_GRAPHCTR, GCRegs[ i ] );

    STI(); /* Enable interrupts */
}

```



```

}

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

void ReleaseFontAccess( void )
{
    static unsigned SeqRegs[4] = { 0x0100, 0x0302, 0x0304, 0x0300 },
        GCRegs[3]   = { 0x0004, 0x1005, 0x0E06 };
    BYTE i;
    /* Loop counter */

    CLI();
    /* Disable interrupts */

    for ( i=0; i<4; ++i )
        /* Load different sequencer registers */
        outpw( EGAVGA_SEQUENCER, SeqRegs[ i ] );

    for ( i=0; i<3; ++i )
        /* Load graphics controller registers */
        outpw( EGAVGA_GRAPHCTR, GCRegs[ i ] );

    STI();
    /* Enable interrupts */
}

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

```

```

void ClearGraphArea( void )
{
    int exchars,                /* Characters to be executed */
        chrow;                 /* Row within the corresponding character */

    for ( exchars = 0; exchars < 256; ++exchars ) /* Loop characters */
        for ( chrow = 0; chrow < CharHeight; ++ chrow ) /* Loop rows */
            *(fontptr+exchars*CHAR_BYTES+chrow) = 0; /* & set to 0 */
}

/*****
*   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 0xFF exists, the system generates an
*                appropriate color code needed for the mikado effect.
*****/

void InitGraphArea( BYTE x, BYTE y, BYTE xlen, BYTE ylen, BYTE map,
                   BYTE gacol )
{
    unsigned offset;           /* Offset in video RAM */
    int         column, chrow; /* Loop variables */
    BYTE        ccode;         /* Floating character code */

    if ( xlen * ylen > 256 ) /* Range too large? */

```

```

printf( "Error: Area larger than the 256-character maximum\n" );
else
{
    if ( CharHeight == 16 )
        SetCharWidth( 8 );
    SelectMaps( 0, map );
    xmax = xlen*CHAR_WIDTH;
    ymax = ylen*CharHeight;
    lenx = xlen;
    fontptr = MK_FP( 0xA000, map * 0x4000 );
    GetFontAccess();
    ClearGraphArea();
    ReleaseFontAccess();

    /*-- Fill graphic area with character codes -----*/

    ccode = 0;
    for ( chrow = ylen-1; chrow >= 0; --chrow )
        for ( column = 0; column < xlen; ++column )
        {
            offset = ((chrow+y)*80+column+x) << 1;
            *(viopttr+offset) = ccode;
            *(viopttr+offset+1) = ( gacol == 0xff ) ? ( ccode % 6 ) + 1 + 8
                                                    : gacol | 0x08;
            ++ccode;
        }
}

/*****
*   CloseGraphArea: Closes graphic area.
**-----**

```

```

*   Input       : None
*****/

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

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

void SetPixel( int x, int y, BOOL on )
{
    BYTE charnum,                 /* Code for character at coordinates */
        linetr,                 /* Pixel line in the character */
        far *bptr;

    if ( ( x < xmax ) && ( y < ymax ) ) /* Coordinates O.K.? */
    {
        /* Yes --> Compute character no. and line */
        charnum = ((x / CHAR_WIDTH) + (y / CharHeight * lenx));
        linetr = CharHeight - ( y % CharHeight ) - 1;
        bptr = fontp + charnum * CHAR_BYTES + linetr;
        if ( on ) /* Set or unset character? */
            *bptr = *bptr | ( 1 << (CHAR_WIDTH - 1 - ( x % CHAR_WIDTH ) ) );
        else

```

```

    *bptr = *bptr & !( 1 << (CHAR_WIDTH - 1 - ( x % CHAR_WIDTH ) ) );
}
}

/*****
*   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
*****/

/*-- Function for swapping two integer variables -----*/

void SwapInt( int *i1, int *i2 )
{
    int dummy;

    dummy = *i2;
    *i2    = *i1;
    *i1    = dummy;
}

/*-- Main function -----*/

void Line( int x1, int y1, int x2, int y2, BOOL on )
{
    int d, dx, dy,
        aincr, bincr,
        xincr, yincr,
        x, y;

```

```

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

    xincr = ( x2 > x1 ) ? 1 : -1;        /* Set X-axis increment */

    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 );
    for (y=y1+1; y<= y2; ++y )
    {
        if ( d >= 0 )
        {
            x += xincr;
            d += aincr;
        }
        else
            d += bincr;
        SetPixel( x, y, on );
    }
}

```

```

else
{
    if ( x1 > x2 )
    {
        SwapInt( &x1, &x2 );
        SwapInt( &y1, &y2 );
    }

    yincr = ( y2 > y1 ) ? 1 : -1;

    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 );
    for (x=x1+1; x<=x2; ++x )
    {
        if ( d >= 0 )
        {
            y += yincr;
            d += aincr;
        }
        else
            d += bincr;
        SetPixel( x, y, on );
    }
}
}

```

```
/* Check X-axes */
```

```
/* x1 > x2? */
```

```
/* Yes --> Swap X1 with X2 */
```

```
/* and Y1 with Y2 */
```

```
/* Set Y-axis increment */
```

```
/* Set first pixel */
```

```
/* Execute line on X-axes */
```

```

/*****
*   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
*****/

```

```

void SetPalCol( BYTE RegNr, BYTE Col )

```

```

{
    union REGS Regs;                /* Processor registers for interrupt call */

    Regs.x.ax = 0x1000;              /* Video function 10H, sub-function 00H */
    Regs.h.bh = Col;                /* Color value */
    Regs.h.bl = RegNr;              /* Register number of attribute controller */
    int86( 0x10, &Regs, &Regs );    /* Call BIOS video interrupt */
}

```

```

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

```

```

void SetPalAry( BYTE *NewColPtr )

```

```

{
    BYTE i;                          /* Loop counter */

    for ( i = 0; i < 16; ++i )        /* Execute 16 entries in array */

```



```

    SetPalCol( i, NewColPtr[i] );          /* Set corresponding colors */
}

/*****
*   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 method included for EGA cards, which do not
*             support interrupt 10H, function 10H, sub-function 07H.
*****/

BYTE GetPalCol( BYTE RegNr )
{
    union REGS Regs;          /* Processor registers for interrupt call */

    if ( CharHeight == 14 )   /* EGA card? */
        return RegNr;        /* Yes --> Cannot read palette registers */
    else                      /* No --> VGA */
    {
        Regs.x.ax = 0x1007;    /* Video function 10h, sub-function 07H */
        Regs.h.bl = RegNr;     /* Register number of attribute controller */
        int86( 0x10, &Regs, &Regs ); /* Call BIOS video interrupt */
        return Regs.h.bh;      /* Palette register contents are here */
    }
}

/*****
*   GetPalAry: Gets contents of 16-color palette registers and places
*             these contents in an array for the caller.
*   -----
*****/

```

```

*   Input   : ColAryPtr = Palette array of type PALARY, into which   *
*               colors are placed                                     *
*****/

```

```

void GetPalAry( BYTE *ColAryPtr )
{
    BYTE i;                                     /* Loop counter */

    for (i = 0; i < 16; ++i )                  /* Execute 16 entries in array */
        ColAryPtr[i] = GetPalCol( i );         /* Set corresponding colors */
}

```

```

/*****
*   Mikado: Applies the functions in this program.                  *
**-----**
*   Input   : None                                                 *
*****/

```

```

void Mikado( void )
{
    typedef struct {                               /* Get coordinates of a line */
        int x1, y1,
            x2, y2;
    } LINEIE;
}

```

```

static PALARY NewCols =
{ /----- 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 */

int      i, j,                /* Loop counter */
first,   /* Array index of most recent mikado */
last;    /* Array index of oldest mikado */
BOOL     clear;              /* Clear mikados */
LINIE    lar[MIKADOS];       /* Mikado array */
PALARY    OldCols;           /* Get old colors */

GetPalAry( OldCols );        /* Get old colors */
SetPalAry( NewCols );        /* Install new color palette */
/*TextColor( 7 );
TextBackGround( 1 );
GotoXY(1,1); */
ClrScr( 0x07 );              /* Clear screen and */
for (i=0; i<25; ++i)         /* fill with characters */
    for (j=0; j<80; ++j )
        PrintfAt( j, i, 0x07, "%c", 32 + (((int) i*80+j) % 224) );

/*-- Initialize graphic area and generate mikados -----*/

```

```

PrintfAt( 27,6, 0x70, "          M I K A D O          " );
SetCursor(27,6);
InitGraphArea( 27, 7, 25, 10, 1, 0xFF );
GetFontAccess();                                /* Get access to font */

clear = FALSE;                                /* No mikados cleared yet */
first = 0;                                    /* Start with first array position */
last = 0;
do
{
    /* Mikado loop */
    if (first == MIKADOS )                    /* Wraparound? */
        first = 0;
    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,        /* and draw it */
          lar[first].x2, lar[first].y2, TRUE );
    if ( ++first == MIKADOS )                  /* Already clear? */
        clear = TRUE;
    if ( clear )                              /* Clear now? */
    {
        /* Yes */
        Line( lar[last].x1, lar[last].y1,
              lar[last].x2, lar[last].y2, FALSE );
        if ( ++last == MIKADOS )
            last = 0;
    }
}
while (!kbhit());                            /* Repeat until user presses a key */
getch();                                    /* Remove key from keyboard buffer */

/*-- End program -----*/

```

```

CloseGraphArea();
SetPalAry( OldCols );          /* Restore old color palette */
SetCursor(0,24);
printf( "\nSystem has reverted to old font.\n" );
}

```

```

/*****
*                               M A I N   P R O G R A M                               *
*****/

```

```

void main()
{
    if ( IsEgaVga() )           /* Is there an EGA or a VGA card installed? */
        Mikado();              /* Yes --> Execute demo */
    else                         /* No --> Program cannot be started */
        printf( "Warning: No EGA or VGA card found" );
}

```