ANALOG DEVICES

AN-330
APPLICATION NOTE

ONE TECHNOLOGY WAY • P.O. BOX 9106 • NORWOOD, MASSACHUSETTS 02062-9106 • 617/329-4700

Animation Using the Pixel Read Mask Register of the ADV47X Series of Video RAM-DACs

by Bill Slattery & Eamonn Gormley

INTRODUCTION

The Pixel Read Mask Register, which is an integral part of IBM's VGA* graphics system, can be used as a hardware-level Pixel Processing Unit. This allows real time motion or animation to be implemented with minimal software overhead. This application note examines the operation and structure of such a pixel processing unit with the pixel read mask register as the central controller. A practical application which uses the pixel read mask register to animate a picture scene is described. A complete listing of the Turbo-C source code is given in the appendix.

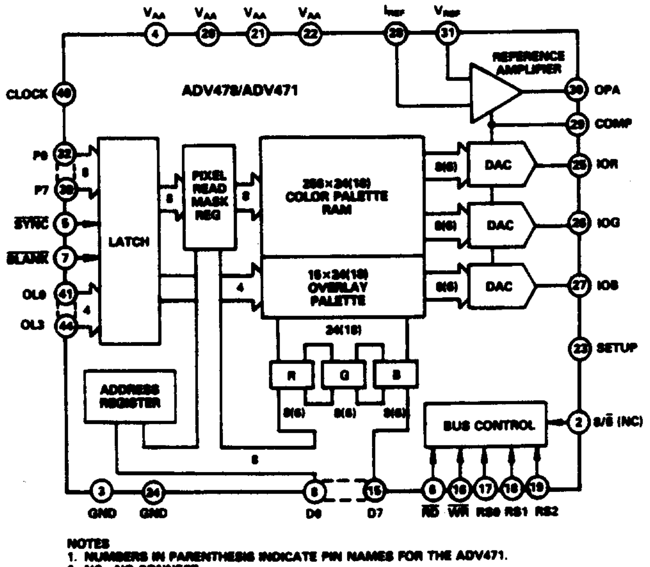
No additional hardware is required for existing VGA graphics systems to implement this application.

VIDEO RAM-DAC

Analog Devices produces a range of video RAM-DACs, which are specifically designed for IBM's Personal System/2* VGA. The range includes the ADV478, ADV471 and ADV476, all of which are monolithic +5 V CMOS video RAM-DACs. These parts are specified over a number of speed grades; 35 MHz, 50 MHz, 66 MHz and 80 MHz. The RAM-DACs are packaged as 44-pin PLCC and 28-pin plastic DIP devices.

The ADV471 and ADV476 each contain a triple 6-bit digital-to-analog converter and a 256 location by 18 bits deep color look-up table. The devices also include an asynchronous pixel input port and bidirectional microprocessor (MPU) port. These devices and the associated control circuitry allow for flexible interface to many graphics systems configurations. The ADV478 differs from the ADV471 only in terms of its color resolution. The ADV478 has a triple 6-bit/8-bit D/A converter with a 256 × 24/18 color look-up table. The color resolution of the ADV478 is user selectable between 6 bits and 8 bits. The higher 8-bit performance can be used with IBM's 8514/A* graphics standard (upgrade on standard VGA). More detailed information on these and other video RAM-DACs can be obtained in the relevant product data sheets.

Built into all three devices is an 8-bit register known as the Pixel Read Mask Register. Figures 1 and 2 are block diagrams of the ADV478/ADV471 and ADV476 which show the Pixel Read Mask Register.



NOTES
1. NUMBERS IN PARENTHESIS INDICATE PIN NAMES FOR THE ADV471.
2. NC=NO CONNECT

Figure 1. ADV478/ADV471 Functional Block Diagram

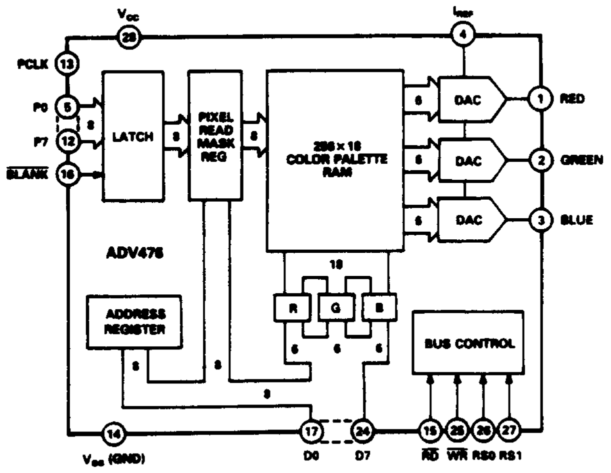


Figure 2. ADV476 Functional Block Diagram

*IBM, VGA, Personal System/2 and 8514/A are trademarks of International Business Machines Corp.

Some of the uses to which the Pixel Read Mask Register can be put include on-screen special effects such as real time animation, flashing objects and overlays.

PIXEL READ MASK REGISTER

The Pixel Read Mask Register is placed in the path of the pixel input stream of data as shown in Figure 3.

The input pixel data stream (P0–P7) is gated with the contents of the Pixel Read Mask Register. The operation is a bitwise logical ANDing of the pixel data. The contents of the Pixel Read Mask Register can be accessed and altered at any time by the MPU (D0–D7). Table I shows the relevant control signals. Under normal operating conditions, this register is loaded with all 1s, i.e., transparent mode.

In a VGA graphics system, the Pixel Read Mask Register is memory mapped and is accessible (read/write) by addressing memory location 36CH.

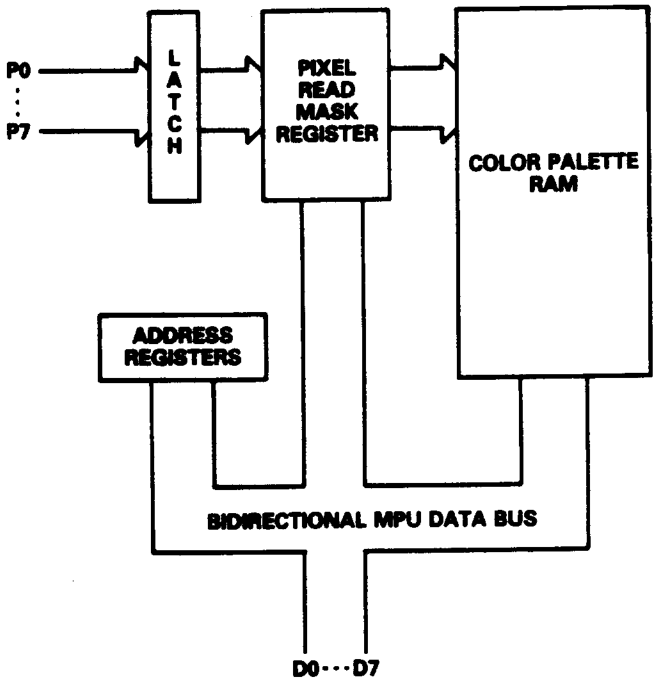


Figure 3. Video RAM-DAC Pixel & Data Ports Showing the Pixel Read Mask Register

RS2*	RS1	RS0	Addressed by MPU
0	0	0	Address Register (RAM Write Mode)
0	1	1	Address Register (RAM Read Mode)
0	0	1	Color Palette RAM
0	1	0	Pixel Read Mask Register

*RS2 is only present on ADV478/ADV471

Table I. Control Input Truth Table for Video RAM-DAC

Figure 4 shows the internal architecture of the pixel input port. The input word Pi, which corresponds to an on-screen pixel location, is ANDed with the contents of the Pixel Read Mask Register, Pm.

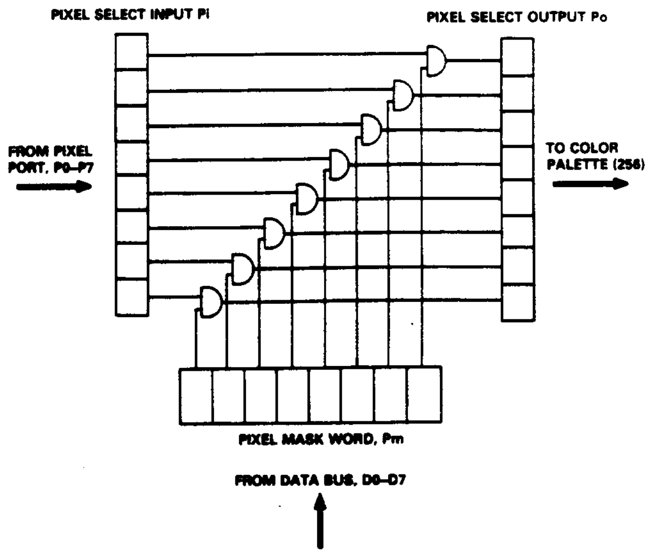


Figure 4. Internal Architecture of Pixel Input Port

The resulting output word, Po, determines which location in the color palette will be assigned to a particular on-screen pixel. Figure 5 shows the logical diagram for this masking operation.

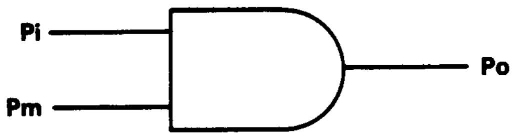


Figure 5. Equivalent Logical Representation of Masking Operation

$$Po = Pi \cdot Pm \quad (1)$$

If Pm = 1, transparent mode, then

$$Po = Pi \quad (2)$$

The pixel stream of data, Po, which arrives at the color palette is a function of both the pixel input stream, Pi, from the Frame Buffer and the contents of the Pixel Read Mask Register, Pm. In the case of an animation application, which will be discussed later in this application note, the rate at which the pixel mask word is changed will determine the motion speed of the scene.

This pixel masking operation can be used to alter the displayed colors without changing the contents of either the video Frame Buffer or the Color Palette RAM. One interpretation of this operation is to consider the pixel input structure of the video RAM-DAC as an on board Pixel Processing Unit.

PIXEL PROCESSING UNIT

The Pixel Input Port (Pi), Pixel Read Mask Register (Pm) and Data Input Port (MPU) within the video RAM-DAC, are the hardware components of this Pixel Processing Unit (PPU). An associated software routine to control the operation is the final element in the complete PPU system. This interpretation enables the color palette to be configured as a multidimensional, paged memory address space, see Figure 6.

In the case of the ADV471 and ADV476 the color palette can be perceived as being broken into an even number of 18-bit color planes instead of just one 18-bit deep color plane. (In the case of the ADV478, each plane is 24 bits deep.)

The palette can therefore be partitioned to produce up to a total of 256 discrete contiguous color memory planes, some of which are shown in Figure 6. A tradeoff, however, must be considered when dividing the color palette into multiple color planes. The number of simultaneously displayable screen colors is inversely proportional to the number of color planes within the color palette. Table II illustrates this relationship. This contiguous configuration is not, however, the sole way of segmenting the memory within the palette. Other non-contiguous configurations including interleaving can be implemented. The choice of memory configuration will be determined by the particular application so as to make most efficient use of the available video memory (Image Frame Buffer and Color Palette RAM).

Number of Color Planes	Number of Simultaneously Displayable Colors
1	256
2	128
4	64
.	.
.	.
256	1

Table II. Simultaneously Displayable Screen Colors versus Number of Color Planes

To operate the PPU, two principal steps must be taken:

1. Load the image data in the correct paged configuration to both the frame buffer and color look-up table, e.g., four frame composite images to the frame buffer corresponding to four discrete planes of color to the palette RAM.
2. Generate the corresponding pixel mask words. These words are individually written to the Pixel Read Mask Register, Pm (at VGA memory location 36CH) and select which of the color planes is to be assigned to the incoming pixel data stream. In the case where four planes are implemented (see Figure 6), four pixel mask words are required.

The overall VGA System Block Diagram showing the PPU and an associated 8-page memory configuration of the color palette is shown in Figure 7.

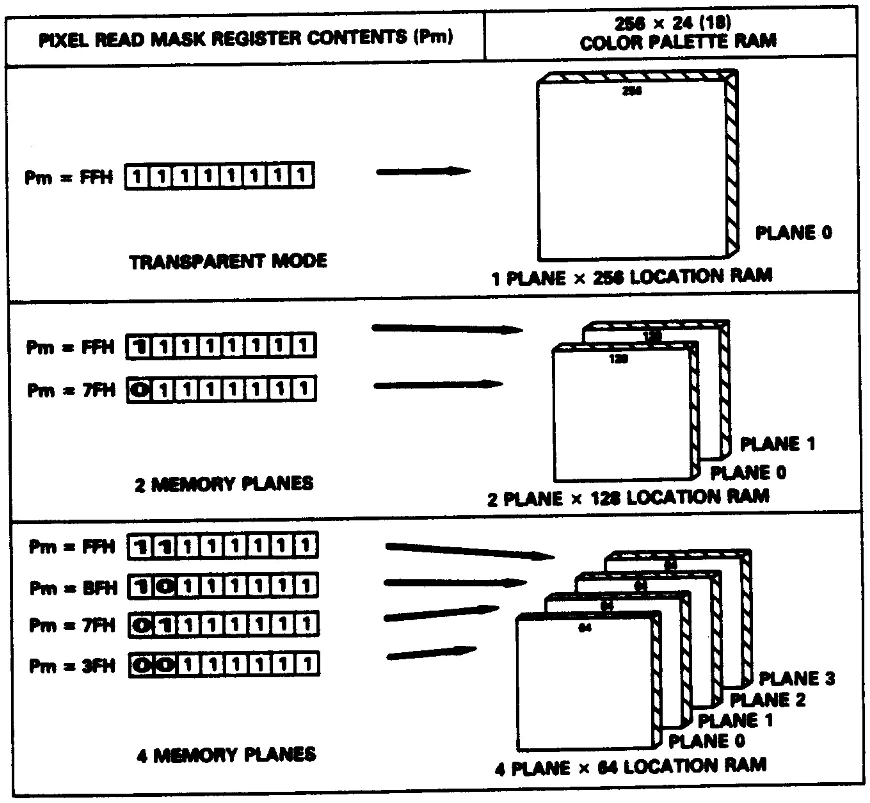


Figure 6. Some Color Palette RAM Configurations Showing Paged Memory Planes and Associated Pixel Read Mask Word (Pm)

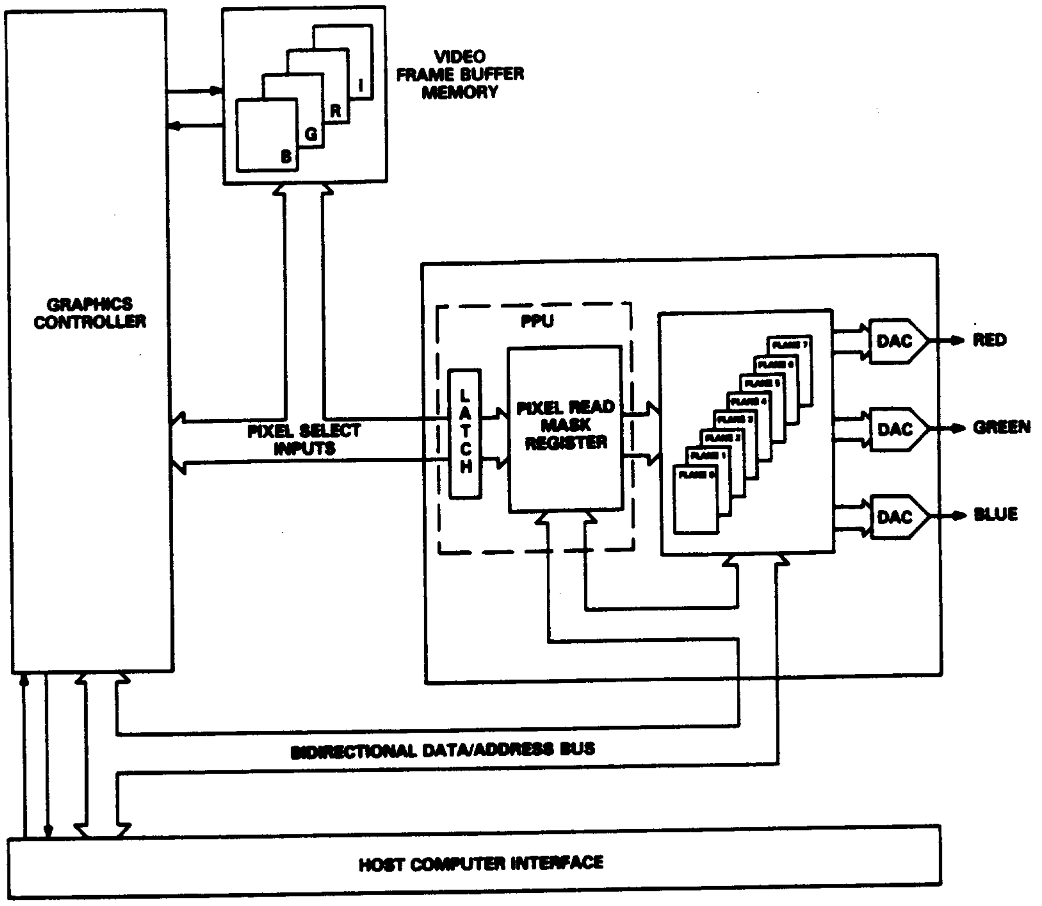
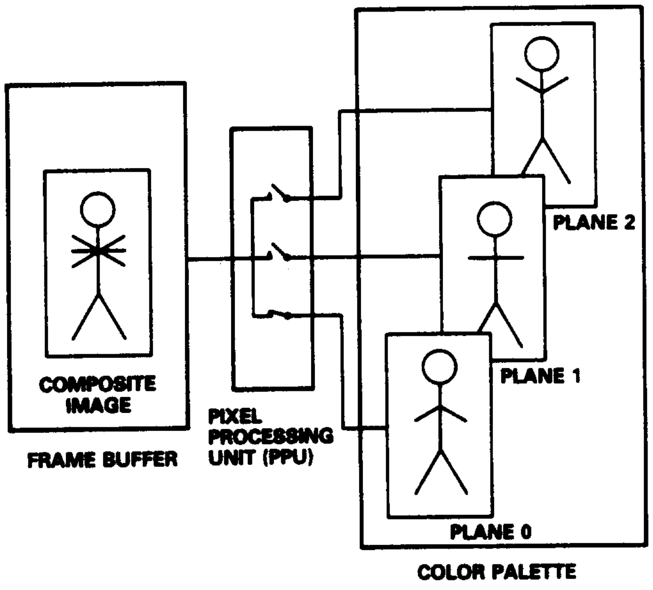


Figure 7. VGA System Block Diagram Showing Color Palette Broken into a Number of Color Planes

ANIMATION

Real time animation using the Pixel Processing Unit is based on the principle that rapidly changing the colors of a stationary object gives the illusion of motion. In other words, a number of similar images or frames, differing only by the relative position of the various colors, displayed in quick succession, can result in motion.

A simple example to explain the idea of animation is illustrated opposite. The animated image consists of three frames; each of the three frames is initially drawn as one composite picture (Frame Buffer image). The color palette contains three discrete, memory blocks or planes of color information, corresponding to three stages of animation. The animation effect in this example is "arm waving" of the cartoon character. By assigning the color planes one by one to the image in the Frame Buffer, the effect of animation can be perceived on the screen. The color plane assigned to the composite image is determined by the PPU which is controlled by the word in the Pixel Read Mask Register. Frame 1 is assigned Color Plane 0, this colors the down arm position in black, while the up and horizontal arm positions take on the background color. Frame number 2 is assigned Color Plane 1, this colors the horizontal arm position in black while the other two arm positions are assigned the back ground color.



Finally, Frame 3 takes on Color Plane 2. This process is then repeated giving the illusion of motion. The rate at which each frame is selected determines the rapidity of the arm waving.

ANIMATION USING THE PPU

This section describes a particular animation example. The scene used in this example consists of traveling space ships and rotating planets. The program which draws the scene and implements the animation is described in the flow diagram of Figure 8. The associated source code, written in Borland's Turbo-C, is given in the Appendix. This application implements 8-stage animation.

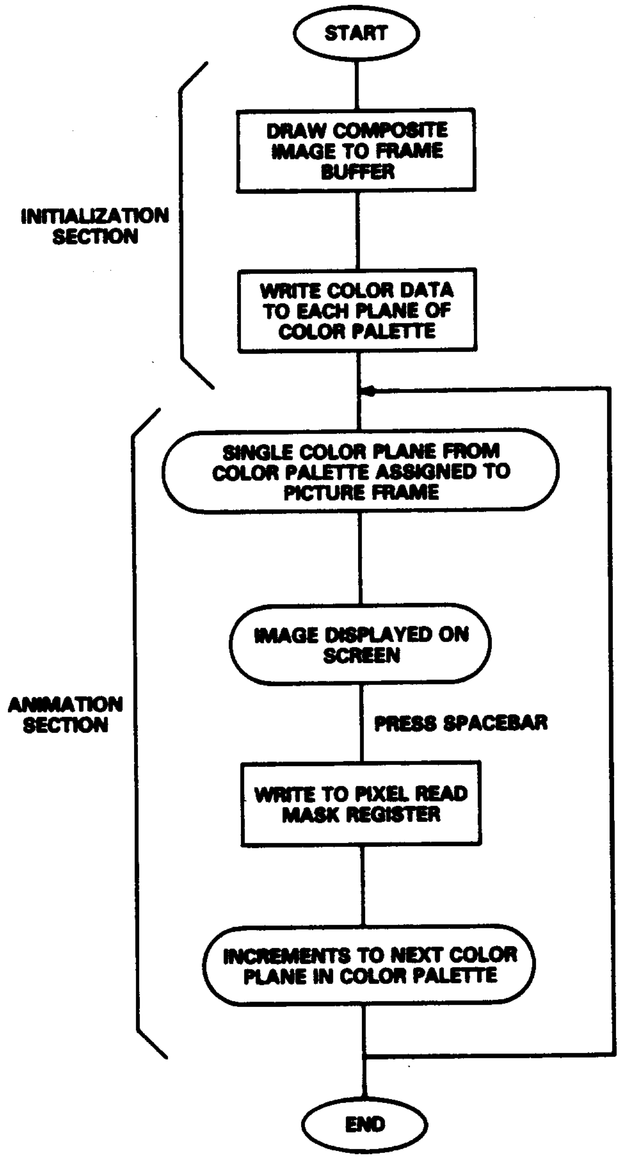

START
DRAW COMPOSITE IMAGE TO FRAME BUFFER
INITIALIZATION SECTION
WRITE COLOR DATA TO EACH PLANE OF COLOR PALETTE
SINGLE COLOR PLANE FROM COLOR PALETTE ASSIGNED TO PICTURE FRAME
IMAGE DISPLAYED ON SCREEN
ANIMATION SECTION
PRESS SPACEBAR
WRITE TO PIXEL READ MASK REGISTER
INCREMENTS TO NEXT COLOR PLANE IN COLOR PALETTE
END

Figure 8. Flow Diagram Representation of Animation Using the PPU

The complete image is drawn to the Frame Buffer. This composite picture contains eight frames of information. The corresponding color planes for each of the eight frames in this composite image, are drawn to the color palette. The color information is arranged in a paged memory format corresponding to that shown in Figure 7. Each of these eight color planes has similar color data; they differ from each other only in terms of the relevant position of the particular colors. For example, Plane 0 could have blue in its first location and color yellow in its second location, while Plane 1 could have the opposite, yellow in Location 1 and blue in Location 2. During the display period, the color palette will only allocate colors to one of the eight frames (i.e., one color plane) at a particular instant. Each plane of color information is mapped to a particular frame within the Frame Buffer. The user-defined value of the Pixel Read Mask Register determines which of the color planes within the palette will be chosen for display at any particular instant. The hex codes, written to the Pixel Read Mask Register Pm, which correspond to each of the color planes (Plane 0 to Plane 7) are listed in Table III.

3C6F	: Address of Pixel Read Mask Register (Pm)	
8FH → Pm	: Plane 0 Selected	Pm = 1000 1111
9FH → Pm	: Plane 1 Selected	Pm = 1001 1111
AFH → Pm	: Plane 2 Selected	Pm = 1010 1111
BFH → Pm	: Plane 3 Selected	Pm = 1011 1111
CFH → Pm	: Plane 4 Selected	Pm = 1100 1111
DFH → Pm	: Plane 5 Selected	Pm = 1101 1111
EFH → Pm	: Plane 6 Selected	Pm = 1110 1111
FFH → Pm	: Plane 7 Selected	Pm = 1111 1111

Table III. Value Written to Pixel Read Mask Register and Associated Color Plane

Pressing the "Spacebar" increments the pixel read mask register corresponding to a jump of 16 locations in the color palette. As there are 16 colors in each color plane, a jump of 16 locations will select the corresponding color in the next highest plane. Continuously pressing the "Spacebar" cycles the incoming pixel stream of data through each of the eight color planes within the palette. This results in the apparent motion or animation of the image.

APPENDIX

C Program for ANIMATION EXAMPLE

Pixel Processing Using Video RAM-DACs
"Rotating Planets & Spaceships"

```
#include <stdlib.h>  /* Turbo C include files */
#include <math.h>    /* these are available under most versions */
#include <dos.h>     /* of C for the IBM & compatibles */
#include <graphics.h>
void palette(int col,int red,int green,int blue);
void plot13(int x,int y,int col);                  /* function definitions */
void mode13();
void circle13(int x,int y,int r,double tilt);
void planets();
void stars();
void line13(int x1,int y1,int x2,int y2,int col);
void triangle();

main()
{
 int gd=0,gm=0,opt;
 union REGS reg;
 detectgraph(&gd,&gm);              /* check for a VGA card */
 if (gd != 9){
  printf("This program cannot find a VGA card installed in this computer.\n");
  printf("A VGA card is necessary to run the tests.");
  exit(1); }

 planets();                         /* do demo */
 reg.h.ah = 0x00;
 reg.h.al = 0x03;
 int86(0x10,&reg,&reg);             /* return to text mode when finished */
}

void mode13()                       /* set up mode hex 13 = decimal 19 */
{                                   /* this is a 256 color mode with */
 union REGS reg;                    /* 320 x 200 pixel resolution */
 reg.x.ax = 0x0013;
 int86(0x10,&reg,&reg);             /* set mode 0x13 */
}

void palette(int col,int red,int green,int blue)
             /* assigns a physical color to a logical color */
{
  union REGS reg;
     reg.x.ax = 0x1010;
     reg.x.bx = col;
     reg.h.dh = red;
     reg.h.ch = green;
     reg.h.cl = blue;
     int86(0x10,&reg,&reg);        /* call bios routine to change palette */
}

void plot13(int x,int y,int col)   /* special plot routine for mode 0x13 */
{
     union REGS reg;
     if(x>=0 && y>=0 && x <320 && y<200){
        reg.x.dx = y;              /* set up registers */
        reg.x.cx = x;
        reg.h.ah = 0x0c;
        reg.h.al = col;
        int86(0x10,&reg,&reg);     /* call bios plot routine */
     }}

void circle13(int x,int y,int r,double tilt)
{                                   /* routine draws a single planet */
```

```
int la,yy;
double ang,oldx,oldy,newx,newy,sintl,costl,rcos,rsin;
for(la = -r; la < r; la++)
{                                     /* routine uses a fairly simple */
 yy = sqrt(r*r - la*la) + 1;          /* algorithm to draw a solid circle */
 line13(la+x,y-yy,la+x,y+yy,14); }

costl = cos(tilt);                    /* set up some variables */
sintl = sin(tilt);
yy = 240;                             /* To draw lines of longitude: */
for(la = r; la >= -r; la-=r/15)       /* draw portions of ellipses */
  {                                   /* and rotate them by tilt radians */
   oldx = x-r*sintl;
   oldy = y+r*costl;
   for(ang = -1.57;ang <1.57;ang+=.195) {
    newx = x+la*cos(ang)*costl+r*sin(ang)*sintl;
    newy = y-r*sin(ang)*costl+la*cos(ang)*sintl;
    line13(newx,newy,oldx,oldy,yy); /* line segment of ellipse */
    oldx = newx;                      /* store endpoints */
    oldy = newy;  }
   yy = (yy==247) ? 240 : ++yy;       /* increment color used */
  }
 for(ang=-1.57;ang<1.57;ang+=.39)     /* draw lines of latitude */
   {                                  /* ie sloped lines */
    rcos = r*cos(ang);
    rsin = r*sin(ang);
    line13(x+rcos*costl-rsin*sintl,y+rsin*costl+rcos*sintl
          ,x-rcos*costl-rsin*sintl,y+rsin*costl-rcos*sintl,15);
   }}

void planets()            /* routine to draw and animate the planets */
{
 int la,lb;

 mode13();
 palette(7,255,255,255);
 printf("       Pixel Read Mask Demo\n");
 printf("     =========================\n");
 printf(" This program contains an animated     picture scene which ");
 printf("is initially drawn  on the screen and then ANIMATED ");
 printf("using   the Pixel Read Mask Register.\n");
 printf("\n  Press the spacebar to draw scene and  hold it down ");
 printf("when scene is ready for   animation. When finished, ");
 printf("press any     other key......");
 while(getch() != ' ');               /* wait for keypress */

 mode13();
 for (la=8;la<16;la++)                /* set up the palette for animation */
   {
    for (lb=0;lb<8;lb++)
    palette(la*16+lb,0,10,63);        /* set planet lines to blue */
    for (lb=8;lb<16;lb++)
    palette(la*16+lb,0,0,0);          /* stars are initially black */
   }
 for (la=128;la<256;la+=17)
    palette(la,63,63,63),             /* define one line on planet to white */
    palette(la+8,63,63,0);            /* and one star to yellow, per frame */

  palette(15,255,255,255);            /* set color 15 to pure white */
  palette(7,20,255,0);                /* color 7 to green */
  palette(14,0,10,63);                /* color 14 used for planet background */

  stars();                            /* draw stars in background */
  circle13(30,30,30,0.9);             /* draw the actual planets */
  circle13(280,35,35,4.0);
  circle13(130,100,70,-0.8);
  circle13(40,240,125,0.5);
  triangle();                             /* draw the spaceship thingy */
  gotoxy(30,21);printf("Space to");
  gotoxy(30,22);printf("animate.");       /* on screen instructions */
  gotoxy(30,24);printf("Other key");
  gotoxy(30,25);printf("to stop.");
   la=143;                                /* 143 = %10001111 */
   do
    outportb(0x3c6,la),                   /* this part does the actual animation */
    la = (la<255) ? la+16 : 143;          /* loop through the palette */
   while((lb = getch()) == ' ');          /* while the spacebar is being pressed */
}

void stars()                              /* routine to plot in the stars */
{
 int la,lb,lc,ld,le,col = 248;
 long q;
 srand(time(&q) % 37);                    /* set up random background */
 for (la=0;la<200;la+=5)  {
    lc = (rand()&0x7)-0x4;
    ld = la;
    le = (rand()&7)+3;
    for (lb=1;lb<320;lb+=le,ld=la+lc*lb/64)
       plot13(lb,ld,col),      /* plot the star */
       col = (col == 255) ? 248 : ++col;
   }}

void line13(int x1,int y1,int x2,int y2,int col)
{                                                 /* this routine draws a line in */
 int la,lb,lc;                                    /* graphics mode 13H */
 if (abs(x1-x2) > abs(y1-y2))  {          /* line longer in x or y direction ? */
   lc = (x2-x1);lb = (x2 - x1 >=0) ? 1 : -1;
   for (la=x1;la!=x2;la+=lb)                      /* loop works out the points on */
   plot13(la,y1+(la-x1)*(y2-y1)/lc,col);          /* the line and plots them */
  }
  else  {
   lc = (y2-y1);lb = (y2 - y1 >=0) ? 1 : -1;
   for (la=y1;la!=y2;la+=lb)
   plot13(x1+(la-y1)*(x2-x1)/lc,la,col);
  }}

void triangle()             /* This routine draws a simple spacecraft-type */
{                           /* object for animation. */
 int la,lb=19,col=248;      /* starting size = 19, color = 248 */
 double tilt=0.5236;        /* starting tilt */

 for (la=200;lb>0;la-=lb,lb--,tilt += .3)
 {                          /* loop to draw 19 objects */
  line13(200+la/2+lb*cos(tilt),la+lb*sin(tilt),
         200+la/2+lb*cos(tilt+2.0944),la+lb*sin(tilt+2.0944),col);
  line13(200+la/2+lb*cos(tilt+2.0944),la+lb*sin(tilt+2.0944),
         200+la/2+lb*cos(tilt+4.1888),la+lb*sin(tilt+4.1888),col);
  line13(200+la/2+lb*cos(tilt+4.1888),la+lb*sin(tilt+4.1888),200+la/2,la,col);
  line13(200+la/2,la,200+la/2+lb*cos(tilt),la+lb*sin(tilt),col);
  col = (col==255) ? 248 : ++col;     /* col = col + 1 until col = 255, when */
 }}                                   /* col returns to zero */
```