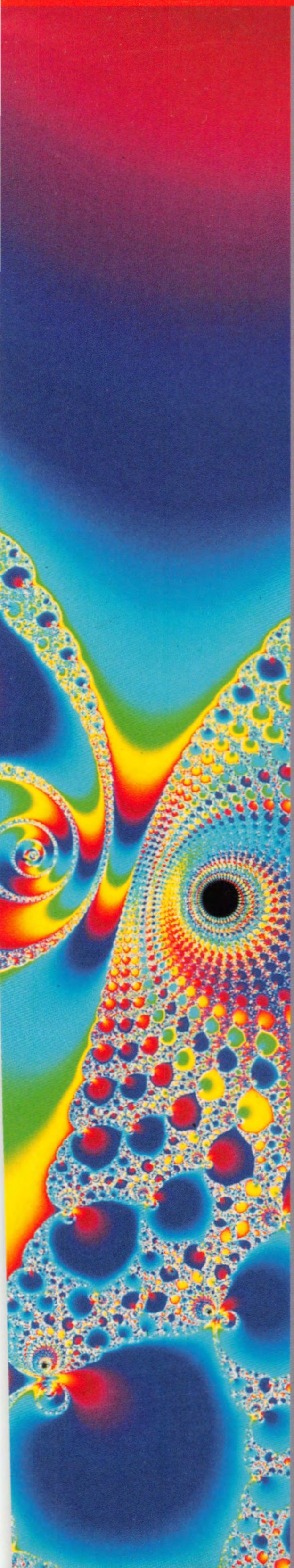
omputing McGraw-Hill



# Computer Animation Programming Methods & Techniques

Julio Sanchez
Maria P. Canton

# Computer Animation

# Computer Animation

## Programming Methods and Techniques

**Julio Sanchez**
*Montana State University, Northern*

**Maria P. Canton**
*Skipanon Software Co.*

**McGraw-Hill, Inc.**

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

**Library of Congress Cataloging-in-Publication Data**

```
Sanchez, Julio, date.
   Computer Animation : Programming Methods and Techniques / Julio
  Sanchez and Maria P. Canton.
   Includes index.
   ISBN 0-07-054964-8
   1. Computer animation. 2. MS-DOS (Computer file) 3. Windows
  (Computer programs) I. Canton, Maria P. II. Title.
  TR897.7.S36 1995
  006.6--dc20                                              94-47025
                                                               CIP
```

1 2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 0 9 8 7 6 5

ISBN 0-07-054964-8

*The sponsoring editor for this book was Jerry Papke, the editing supervisor was David E. Fogarty, and the production supervisor was Suzanne W. B. Rapcavage.*

*Composition and line art by Skipanon Software Co.*
*Printed and bound by Quebecor / Martinsburg.*

# Contents

# Preface

This book is a presentation of practical methods and techniques for animation programming on the personal computer. It includes code samples of the fundamental device drivers and primitives for VGA, SuperVGA, and XGA video systems. It is also a tutorial on low-level animation programming which can be used both as a self-teaching tool and as a general technical reference in the field.

The book presents a low-level approach. The authors believe that in the field of PC animation the low-level approach is more a necessity than an option. This statement seems to contradict the implications of many animation titles currently on the shelves; however, it is our experience that the immense majority of animation programmers in the PC environment are forced, sooner or later, into low-level programming by considerations of performance and control. Every experienced programmer knows that a statement such as "high-performance animation in Basic" or "lighting-fast animation routines in Pascal" is an oxymoron.

Unfortunately the same can be said about animation programming using operating system services. The typical animator is usually engaged in squeezing the last performance drop out of the hardware. There is nothing to waste in the complications, inefficiency, and control limitations of the interface calls of operating system services. It is under DOS, where the code has unrestricted and direct access to the CPU, the video hardware, and the other programmable components, that the best animation results are obtained. Since OS/2 and Windows NT can multitask DOS programs, it is possible to develop DOS applications that execute in these environments with few penalties. In the near future there will be few reasons for animation programming at the operating system level.

This does not mean that animated programs must be coded entirely in assembly language, although this is the case regarding many of the best present-day examples. Once the fundamental primitives and device drivers are available to the code, it is possible to compose the program in a high-level language, achieving acceptable performance by referencing core processing routines coded in assembler. Since this book includes a listing of these core device drivers and primitives, it can be of assistance to the animation programmer working in a high-level environment.

Before attempting to move objects on the video screen, the animation programmer must gain elementary skills in the field of computer graphics. In fact, animation is a specialty field of graphics programming, which implies that many animation techniques require knowledge of computer graphics. For this reason this book includes enough about PC graphics programming to hopefully make the described animation techniques more understandable and useful. On the other hand, this book is not a tutorial in computer graphics and often assumes elementary knowledge in graphics programming. The reader in need of graphics programming information can consult one of our titles on this subject: *Graphics Programming Solutions* and *High Resolution Video Graphics*, both published by McGraw-Hill.

## Organization

The book is divided into three parts: Part One (Animation Fundamentals) includes Chapters 1 through 4. This first part serves as a general introduction to computer animation and to the supporting elements in computer graphics. Part Two (Animation Programming) includes Chapters 5 through 10. This part is a tutorial on VGA, SuperVGA, and XGA graphics. It includes a detailed analysis of the architecture of these systems as it relates to graphics and animation programming. In Part II we also develop the fundamental device drivers and primitives that are necessary to the animation code. Part III (Animation Techniques) describes the methods and techniques most often required by the PC animator. It also contains programming examples.

## Conflict of Interest

The authors believe that it is unethical for a writer to hold back information or to condition the use of the software furnished in a text to copyrights or to the payment of additional fees. Software developers have legal rights to secrecy, to copyright privileges, and to sell their products on the marketplace. However, in our opinion, a writer's loyalty is to the reader and it would be a conflict of interest to write a text while influenced by commercial considerations regarding the material discussed. Therefore, the software developed by the authors and listed in this book can be copied freely, used as furnished, or modified by the reader, without crediting the authors or paying any additional charges. This statement does not include the copyrights on the book itself, nor does it refer to Shareware software, both of which are subject to other rules.

## Acknowledgments

In our years of teaching at Montana State University, Northern, we have been fortunate to interest some students in the fields of graphics and animation programming. In preparing this book we have had access to code and commercial programs developed by students Dale Niemeyer and David Oard. We thank them for this kindness and wish them luck in their professional endeavors.

The authors would also like to thank the friends and associates who provided advice, support, and assistance in this project. At McGraw-Hill, Jerry Papke, David Fogarty, and Gemma Velten have been involved in the production of this book.

At Montana State University, Northern, we owe thanks to Dr. Karen LaRoe, Vice Chancellor for Academic Affairs, Dr. Richard Fisher, Director of the Great Falls campus, and Virgil Hawkinson, Division Chair, for their continued support of our writing projects. Our colleagues Kevin Carlson, Wes Tucker, Roger Stone, and Jay Howland have also made us feel their enthusiasm. Our thanks also go to Sharon Lowman.

*Great Falls, Montana*

Julio Sanchez
Maria P. Canton

## SOFTWARE ONLINE

The source code listed in the text can be downloaded from the Montana State University, Northern BBS, in Havre, MT.

The BBS phone number is (406) 265-4184

The file is a self-extracting archive named COM_ANI.EXE.

# Computer Animation

Part

1

# Animation Fundamentals

Chapter

1

# The Dynamics of Computer Graphics

## 1.0 Digital Simulation of Movement

Computer animation can be defined as the simulation of movement or of lifelike actions by the manipulation of digital objects. The notion of *digital simulation of movement* is the core of this definition; however, as Magnenat-Thalmann and Thalmann have pointed out, computer animation can exist without the simulation of movement (see Bibliography), for example, in morphing (change of one object into another one) and in transformations produced by changes in color or lighting. In any case, this *simulation of life* has added an exciting dimension to computer graphics and to graphics programming, so that today, computer animated imagery is frequently found in applications related to art, science, and technology.

In the simulation of movement the computer can play two different roles: it can serve as an assistant in the creation of imagery destined for display in other media, or it can itself be the destination of the animated action. In the first case we speak of computer-assisted animation. An example of computer-assisted animation is a transformation called *morphing*. In this case the animator inputs two images: one of the face of a man and another one of a wolf face. The machine is programmed to generate a set of intermediate drawings that gradually change the first image into the second one. All the consecutive images generated by the computer, as well as the original ones, can be stored in video tape or film. When the stored images are played back in sequence, the man's face appears to gradually change into a wolf. This interpolation technique is now a common special effect in television and motion pictures.

But perhaps a more familiar notion of computer animation is that in which the computer is both the generator and the display instrument. This is particularly true in microcomputers, which have been used seldom commercially in animation-assistant roles due to their limitations in storage and processing power. We speak of real-time animation when the computer is the tool, as well as the media. In this sense the imagery is generated and displayed in the user's own time frame; there is no image storage for later playback. The animation is shown "live" as it happens. Often the user interacts with the system to determine or modify the result. For example, a flight simulator program consists of animated imagery of the aircraft's cockpit and of the pilot's view thorough the windshield. The actions performed by the user in the simulated flying of the digital aircraft determine changes in the cockpit instruments and in the landscape seen through the windshield. In this case all the changes required for the animation take place in the user's (pilot) time frame.

In this chapter we describe the general principles, theory, and applications of computer animation. The material serves both as an introduction and as general background for the rest of the book. In preparing it we have relied heavily on the books *Computer Animation: Theory and Practice* and *New Trends in Animation and Visualization*. Nadia Magnenat-Thalmann and Daniel Thalmann are the authors of the first title and the editors of the second one. (See Bibliography.)

## 1.1 Conventional Animation

Conventional animation refers mostly to the techniques used in two-dimensional cartoons often associated with Walt Disney, Hannah-Barbera, and others. The method consists of photographing a series of progressive drawings. The photographs are typically developed as color transparencies and animation achieved by successively projecting the transparencies on the screen. Although microcomputers are not used often in the production of cartoons, the technique is interesting to the graphics programmer since similar methods can be applied to real-life animation problems.

### 1.1.1 Historical background

In 1831 a Frenchman named Joseph Antoine Plateau was able to create the illusion of movement by means of a machine which he called a *phenakistoscope*. The device consisted of a disk with a series of progressive drawings and windows. When the disk was rotated the viewer would see the drawings in rapid sequence, which created an illusion of movement. Three years later an Englishman named Horner modified the phenakistoscope into a device which he called the *zoetrope*. The zoetrope consisted of drum with drawings on its inner walls. A series of slits allowed the viewer to see the different drawings as the drum rotated. Emile Reynaud, another Frenchman, further refined the zoetrope by replacing the viewing slits with mirrors. This device was named the *praxisnoscope*.

Emile Reynaud founded the first movie theater in 1892. It was located in Paris and called the *Theater Optique*, although the first animated film was not produced until 1906. By 1913 several American companies were regularly producing cartoons for the thriving motion picture theaters. *Felix the Cat*, by

Pat Sullivan, is possibly the best known cartoon character of this era. Walt Disney, who is usually considered the father of animated cartoons, produced a *Mickey Mouse* film in 1928. This film was the first one to incorporate sound. *Donald Duck* and other characters followed shortly thereafter. *Snow White and the Seven Dwarfs* was the first feature film-length cartoon.

### 1.1.2 Cartoon Animation Techniques

Computers are playing an expanding role in the commercial production of cartoons. Their use includes the coloring of drawings as well as the generation of intermediate images, an operation called *in-betweening* or *tweening*. Drawing and coloring and in-betweens are tedious and time-consuming operations when performed by hand. The organizational elements in the production of an animated cartoon can be seen in Figure 1.1.

The story told in the animated cartoon is developed in three progressively refined steps, shown in Figure 1.1. The *synopsis* is a short summary of the story, usually in less than one page. The *scenario* describes the story more completely and details the characters and the scenery. The *storyboard* consists of a series of drawings and captions that capture the most important moments depicted in the film.

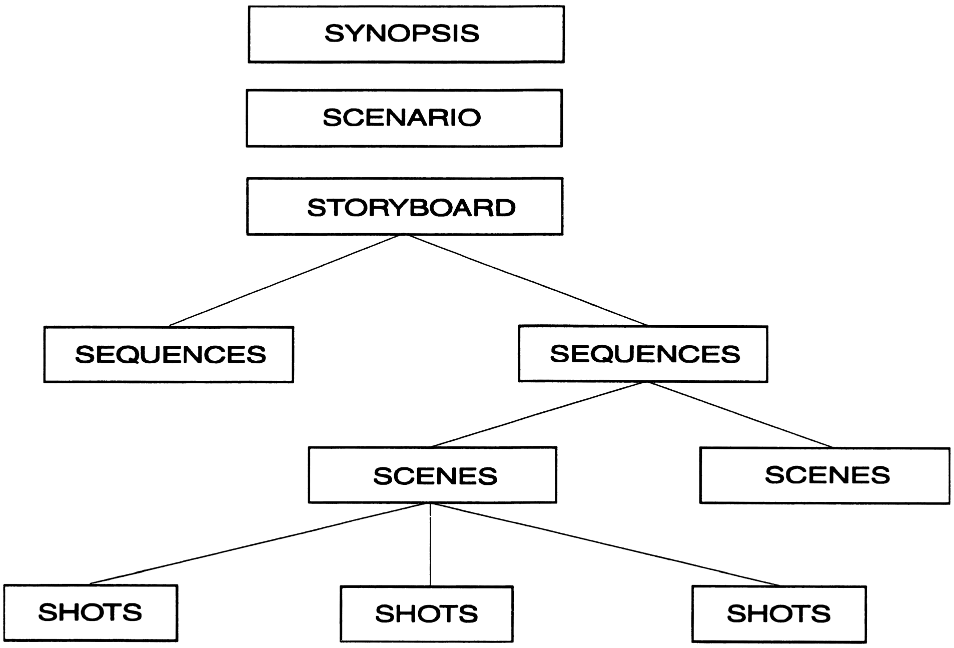


Figure 1.1 *Production Steps for an Animated Cartoon*

From the storyboard it is possible to derive the film *sequences*. Each sequence refers to a film action and consists of one or more *scenes*, typically associated with a particular location and one or more characters. The units of cartoon execution are the individual *shots* that compose each scene. The production of each animated scene is performed by artists called *animators* who lay out, design, and draw the key images in each scene. At this time the sound track for the cartoon must have already been defined, since the motion of the animated figures takes place in relation to dialog and music.

In the production of the actual drawings the artists use as reference two key positions, called frames. For example, Figure 1.2 represents the drawings used in a scene for an animated cartoon in which a dagger appears to travel from the hand of an imaginary thrower to an imaginary target. The key frames are the start frame and the end frame shown in Figure 1.2. The drawings that are necessary to animate the movement between both key frames are the in-between frames. In conventional animation in-betweening is a routine task usually performed by assistants to the animators.

In Figure 1.2 we have drawn three in-between frames. In reality the number of progressions between the start frame and the end frame of a sequence depends on the time assigned to the frame and the display rate. For example, if the animation of the clock sphere in Figure 1.2 is to take 1.5 seconds and the display rate is of 24 frames per second, then 36 frames are required for the animation, of which 34 are in-between frames.

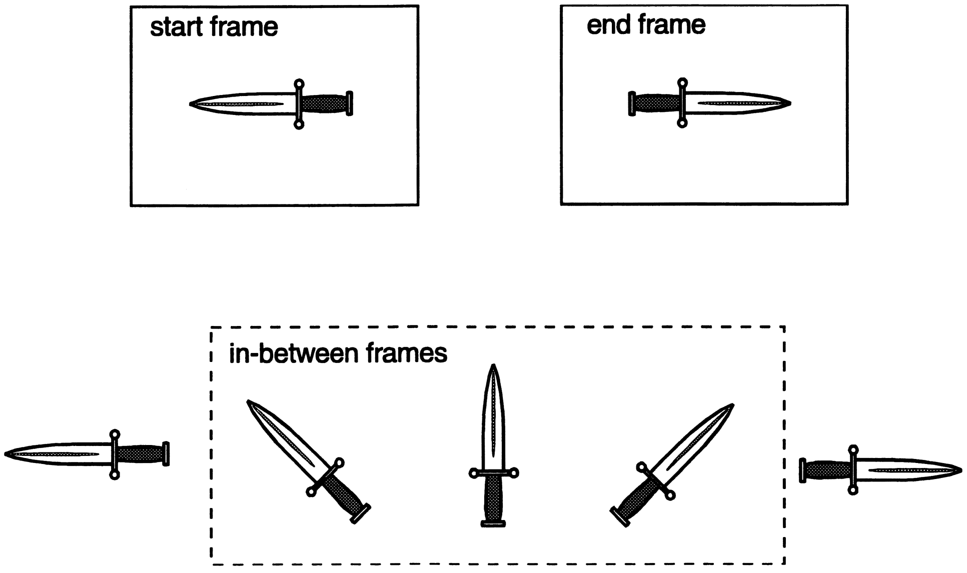


Figure 1.2 *Progressive Drawings in Cartoon Animation*

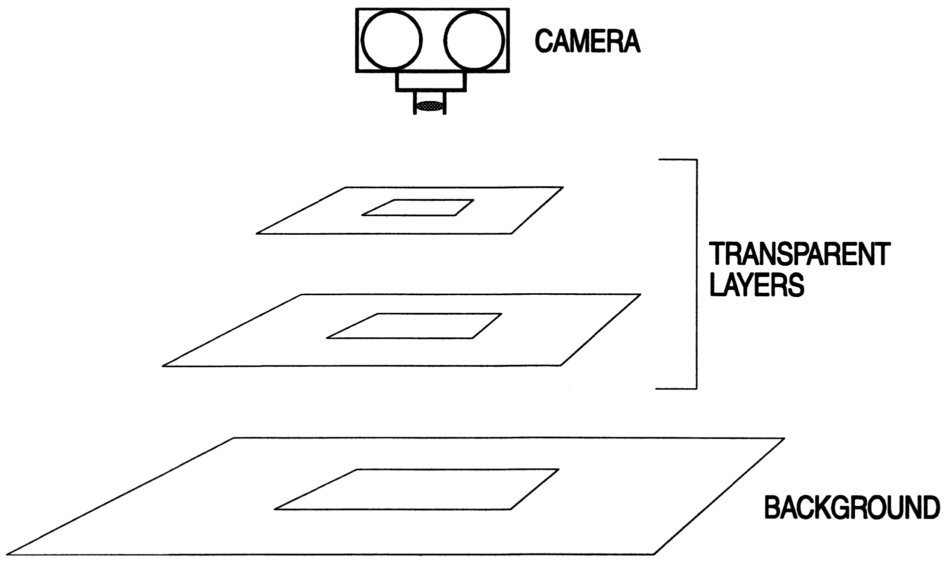


Figure 1.3 *Diagram of a Multiplane Camera*

## 1.1.3 Photographic Manipulations in Cartoon Animation

In addition to the progressive drawings that simulate movement, cartoonists can enhance the animation by means of photographic manipulations. The drawings for cartoon animation are made on a transparent plastic film. Therefore the clear portions of the drawing are invisible to the camera. The equipment used in the production of cartoons is a specialized motion picture camera called a *multiplane*. The animation surface consists of several glass layers at varying distances from the camera lens. Figure 1.3 is a schematic diagram of a multiplane camera.

The multiplane camera is used in creating various special effects. For example, the camera can be moved horizontally to *pan* an image or moved along the optic axis to enlarge or reduce the apparent size of an object (*zooming*). An effect called *spin* is created by rotating the camera. Several *fade* and *dissolve* effects are used in providing a soft transition between scenes. The *fade-in* is a progressive transition of the image from black and the *fade-out* a transition to black. The fade-in is typically used at the start of a scene and the fade-out at the conclusion.

In multiplane animation the image is separated into several elements according to their distance from the viewer. For example, in animating the scenery visible from a moving train it is possible to divide the image into several strips, as shown in Figure 1.4.

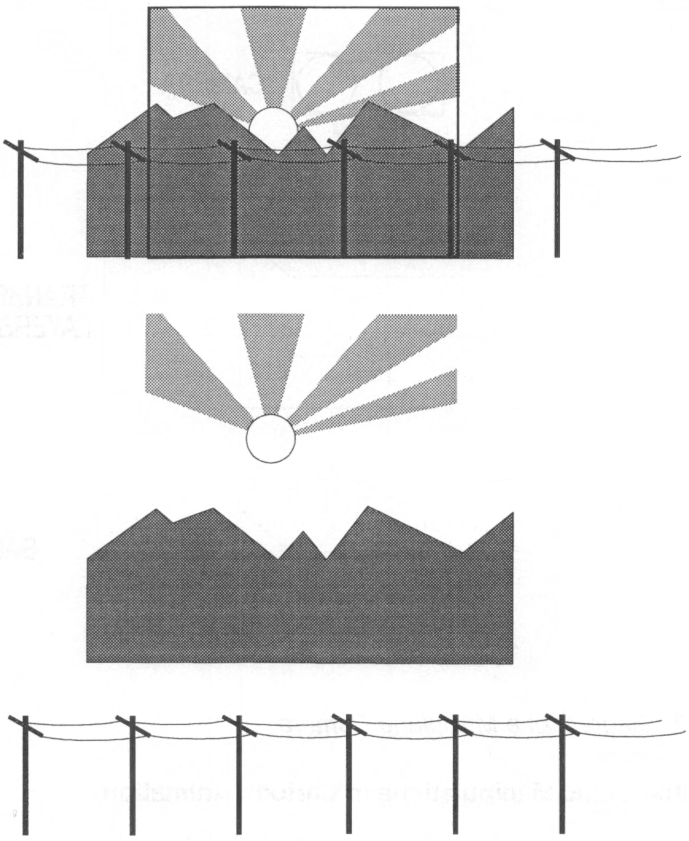


Figure 1.4 *Assembly of a Multiple-Plane Image*

The landscape in Figure 1.4 is animated by moving the three image strips at different rates under the multiplane camera. In this case the image strip that depicts the rising sun remains stationary. The strip depicting the mountain range is moved slowly under the lens while the strip with the telephone poles is moved at a faster rate. The resulting animation would simulate the apparent movement of these objects as viewed by an observer on a train or automobile. Notice that the length of the images is proportional to their rate of movement during animation. Also that more image planes could be used to enhance the realism. The multiple-plane animation technique is quite suited to computer animation, as will be shown later in the book.

## 1.2 Computer Animation

Regarding animation the computer can assume one of two roles: it can assist in the creation of animated imagery (computer-assisted animation) or it can both generate and portray the animated action (real-time animation). The most time-consuming and tedious task of cartoon animation is the generation of the many intermediate images required by the process (tweening). The computer plays the following assistant roles:

1. During the drawing stage the computer is used to scan and digitize image elements and to create drawings or parts of drawings by means of draw or paint software.
2. In the animation process the computer is used to generate in-betweens and to color drawings.
3. During the photography stage the computer controls the multiplane camera and assists in the creation of special effects.
4. In the production stage the computer is used in editing and in adding sound to the animated film.

This list is by no means final. Every day animators find new uses for computers, new technologies are developed which create novel possibilities and applications in animated graphics. Computer technology is being used in the creation of spectacular special effects based on the digitization of screen objects, which are later manipulated by the software. Original efforts in this type of computer-assisted animation are found in the film *TRON*, produced by Walt Disney Studios, as well as in *Return of the Jedi*, by Lucasfilm. In recent years animation by image digitization has become the rule, rather than the exception. It can be found often in science fiction, action, and even conventional films.

### 1.2.1 Animating in Real Time

Real-time animation is found in arcade machines, simulators and trainers, electronic game machines such as those manufactured by Nintendo and Sega, and in interactive programs mainly intended for microcomputers. In real-time animation the computing machine serves both as image generator and as display media.

Animation is based on the physiological fact that the image of an object perceived by the human eye persists in the brain for a brief period of time after the object no longer exists in the real world. This phenomena, called *visual retention*, is related to the chemistry of the retina and to the structure of cells and neurons in the eye. Smooth animation is achieved in cinematography and television by consecutively displaying images at a faster rate than the period of visual retention. This operation, by which a new image replaces the old one before the period of retention has expired, creates in our minds the illusion of a smoothly moving object.

The period of visual retention is a few hundreds of a second. The critical image update rate for smooth animation has been determined to be between 22 and 30 images per second. Modern day moving picture films are recorded and displayed at a rate of 24 images per second. Commercial television takes place at a slightly faster rate. In general, the threshold rate, subject to individual variations, is estimated at 18 images per second. This means that if the consecutive images are projected at a rate slower than this threshold, the average individual perceives a certain jerkiness in the animation. On the other hand, when the image rate equals or exceeds the threshold, our brains merge the images together and we perceive a smoothly animated action.

If we assume that computer animation must take place at an image rate of approximately 20 images per second, then each image must be updated and displayed in a maximum period of one-twentieth of a second. Furthermore, many forms of animation require that the old image be erased from the display before a new one is drawn; otherwise the animation would leave a visible track of objects on the video display. For this reason the image update sequence is a series of redraw, erase, redraw operations, which means that the critical display rate must be calculated from one redraw cycle to the next one. Consequently, the allotted time for the redraw or the erase operation is one-half the display rate, in this case one-fortieth of a second.

These constraints determine that computer animation is often a battle against time: the program is allowed a limited interval in which to update and redraw the image. The animation programmer resorts to every known trick and stratagem in order to squeeze the maximum performance while executing the image update and the display operations. However, quite frequently, even the most imaginative programming cannot overcome the system's limitations. In this case the result is a bumpy and coarse real-time animation that is but a remote likeness of cinematography and television.

As microcomputer systems and video display hardware become more efficient and powerful, the possibilities of real-time animation expand. In our tests we have found that in a 486-based microcomputer with XGA display hardware it is possible to smoothly animate an image that is 40 times larger than the one that could be handled in a 286-based VGA system. On the other hand, commercial software products must aim at the largest possible group of potential customers. This means that the designers of animated programs often aim at ensuring satisfactory execution even in the more primitive systems. For this practical reason, animated programs that squeeze the maximum performance out of state-of-the-art systems are not readily available, since their customer base would be considerably restricted.

### 1.2.2 Frame-by-Frame Animation

Many of the techniques used in computer-assisted animation can be described as frame-by-frame operations. In frame-by-frame animation the computer generates the required images, which are recorded or stored for playback at a later time. This playback can take place in the same machine that generated the image set or in another media. For example, a computer can be used to manipulate the image strips in Figure 1.4 so as to generate a set of 100 progressive pictures. As the images are generated, they are recorded in video tape. When the image set is complete, the animation can be viewed by playing back the video tape. Alternatively the images can be stored in computer memory or disk and played back in the same machine that generated them. In either case the animation is less demanding of machine processing power since the image creation step need not take place in real time. However, if the computer is used to play back the image set, then this operation is subject to the real-time constraints mentioned in the preceding section.

### 1.2.3 Interactive Animation

Interactive animation refers to computer objects that are moved at the user's desire. At present, the most common interactive devices in microcomputers are the keyboard and the mouse, although joysticks and other devices are often used with the more sophisticated games and simulations. In general, the notion of interactive animation includes any technology in which the user exercises some level of control over computer-animated action. By today's standards the ultimate level of interactive animation is called *virtual reality*, a topic discussed in Section 1.4.5.

### 1.2.4 Random or Unpredictable Elements in Animation

Conventionally, the computer simulation of movement is based on programmable or predictable stages. In this manner, the cartoon animator knows beforehand (from the storyboard) all the actions and interactions that will be portrayed in the final rendition. Even in most implementations of virtual reality, every result can be predicted from the user's interaction with the device. Therefore we can say that the system is, by nature, deterministic.

However, many natural systems are of a different nature. Biology students often observe that colonies of bacteria developing in identical media show different patterns of growth. This is due to the fact that in a complex biological system many development factors cannot be determined a priori. In other words, random or unpredictable elements often influence the development of a biosystem. Some modern geologists claim that the disappearance of the dinosaurs was caused by the collision of an asteroid with the earth. If this hypothesis is true, then a small change in the trajectory of the asteroid would have made it miss our planet. Consequently, the evolution of life on earth would have followed an entirely different path.

We have used the terms *random* or *unpredictable* regarding biosystems due to the fact that the preference of one or another term would imply a philosophical judgment. For example, during reproduction, the genes in the male and female chromosomes combine to form the genetic structure of the offspring. If these genes combined according to fixed rules, all siblings would be identical. This is certainly not the case; furthermore, we have no way of knowing beforehand the offspring's exact genetic makeup. Therefore, the mechanics of gene exchange during reproduction can be considered as a random action.

Statistics often serve to describe the unpredictable behavior of a biosystem. For example, in the above-mentioned gene exchange it is often possible to determine, according to their location in the chromosome, that certain genes are more or less likely to be transmitted. However, anything less than absolute certainty implies randomness or unpredictability. If a computer were to simulate the reproduction of a biosystem, it would have to take into consideration these random or unpredictable factors.

## 1.3 Motion Control Techniques

If computer animation is roughly equated with the screen simulation of movement, the methodology for producing the animated effect can be described as a set of motion control techniques. In this respect Allan and Mark Watt, in their book *Advanced Animation and Rendering Techniques* (see Bibliography), refer to procedural, representational, stochastic, and behavioral as the main categories of the animation hierarchy.

From a programmer's viewpoint, animation is implemented by applying one of many low-level methods of motion simulation and control. Some of these methods have been passed on by cartoon animators, while others are digital in nature and, therefore, unique products of the computer environment.

The computer animator is confronted by many limitations and constraints. The most common approach is based on the axiom "whatever works, works." Very often the animation is produced by means of mathematical transformations on the parameters that define one or more screen images. Since movement is a function of time, the laws of physics are often taken into account. For example, in representing a falling object the animator may use the formula that expresses acceleration in a gravitational field to determine the rate of in-betweening that most naturally represents the action. On the other hand, artistic considerations could determine an intentional variation from the physical laws of motion. J. E. Gomez mentions in an article titled "Comments on Event Driven Animation" that the animator is not constrained to obey physical laws; Wily Coyote walks on air for a few seconds before beginning to fall.

### 1.3.1 Tweening and Morphing

The cartoon animator proceeds from two key positions, known as frames, and creates a set of in-between drawings (see Figure 1.2). The entire sequence is photographed and projected to create an illusion of movement. The depiction of animated action by creating and projecting a set of in-between drawings is often called *tweening*; in this sense the intermediate drawings are the *tweens*. Computer animators have successfully borrowed the tweening technique from cartoon animators. Furthermore, in a computer environment the machine can often aid in the creation of the in-between frames by performing geometrical transformations on the key frames.

The tweening required for representing the flight of the dagger shown in Figure 1.2 can be obtained by rotating and translating the start frame. If the image of the dagger is stored in a specific manner, then the animation is produced by mathematical manipulations of a single image file. In Chapter 2 we begin discussing the storage of graphics images in data structures that permit their mathematical manipulations at display time. This subject is revisited on many occasions throughout the book.

Another technique that originated in cartoon animation is called *morphing*. The term relates to the notion of metamorphosis: a transformation in shape, form, or substance that takes place by biological change or by magic and sorcery. Morphing techniques are being extensively used in motion pictures. We are all familiar with the image of an actor or actress transforming into a wolf or a cat. Figure 1.5 shows the morphing of a circle into a square.

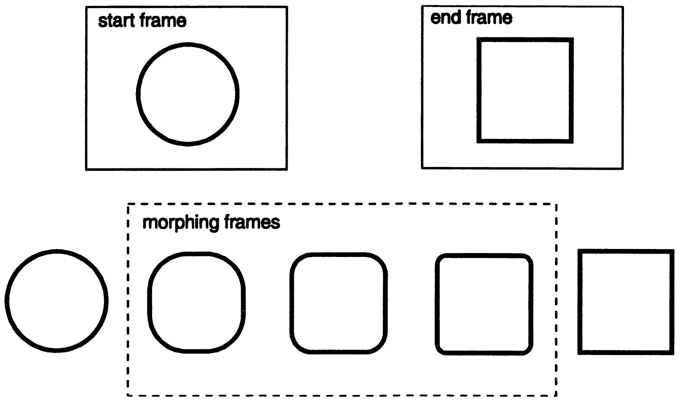


Figure 1.5 *Morphing Animation*

### 1.3.2 Path-of-Motion Calculations

The rules for path-of-motion calculations in animation depend on the image file encoding and on the transformation to be performed. If the coordinate points that define the image are stored in matrix form, then it is possible to perform certain transformations by means of matrix arithmetic. For example, a translation transformation consists of adding a constant value to each coordinate point that defined the object, while a rotation transformation consists of moving all of the object's coordinate points along circular arcs with a common center. Figure 1.6 shows the rotation transformation of the dagger depicted earlier in this chapter.

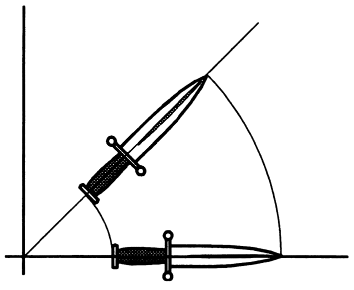


Figure 1.6 *A Rotation Transformation*

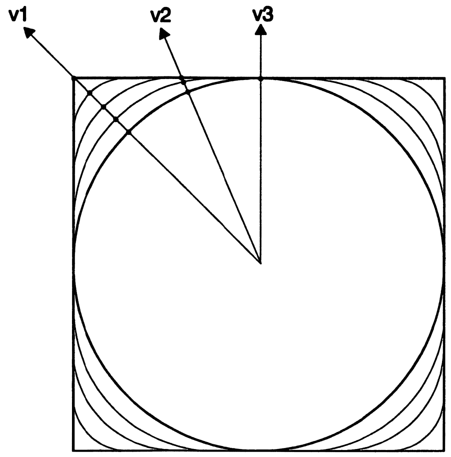


Figure 1.7 *Path of Motion in a Morphing Transformation*

As in tweening, some morphing transformations can be assisted by manipulations of the image file. On the other hand, in morphing, the intermediate frames are determined according to different rules than in tweening. For example, the morphing transformation of a circle into a square shown in Figure 1.5 cannot be made by simple rotation and translation, as is the case in the tweening shown in Figure 1.2. Figure 1.7 shows the path, along a 45-degree vector, that a point on the circle would follow in the process of morphing into a square.

In Figure 1.7, points along different vectors follow different paths. For this reason, morphing usually requires more complicated processing than simple geometrical transformations. Notice that the path of motion along vector **v1** in Figure 1.7 requires three intermediate steps in the transformation of a circle into a square. Along vector **v2** only one intermediate step is necessary, while there is no motion along vector **v3**.

Path-of-motion calculations in tweening and morphing can be rationalized and simplified by using straight lines to approximate geometrical curves. A polygon can be used instead of the circles and curves in the morphing transformation of a circle into a square. This approximation is shown in Figure 1.8.

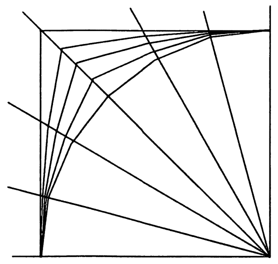


Figure 1.8 *Polygon Approximations in Morphing*

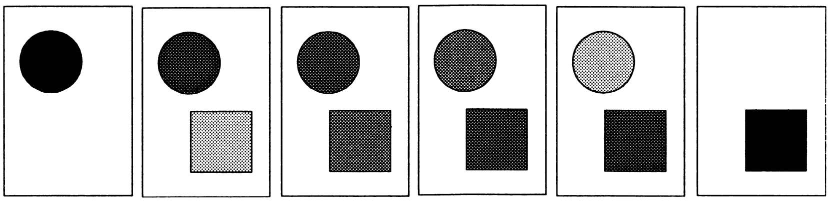


Figure 1.9 *Simultaneous Fade-in and Fade-out*

### 1.3.3 Color-Shift Animation

Objects on the computer screen are furnished with display attributes. In color video systems one category of attributes is the object's color. The animator can manipulate the color attributes of screen objects to create the illusion of movement or change. One common application of this technique is in *fading*. An object or scene is faded-in when its color is progressively changed so as to make it slowly appear on the screen. A fade-out takes place when the object or scene is made to slowly disappear from the screen. Fade-in operations are typically used in cinematography at the beginning of a scene and fade-out at the end of a scene. A cross-dissolve operation takes place when one scene or object is faded-out while another one is faded-in. Figure 1.9 shows the simultaneous fade-in of a rectangle and fade-out of a circle.

In some computer system fade operations can be implemented by progressively changing the hue or saturation of one or more objects or of the entire scene. A screen fade-out can be accomplished by progressively increasing the white saturation of all the objects until the entire screen is white. In some systems (including IBM video systems) the fades can be performed by modifying the color palette itself, instead of the color attributes of individual objects. Palette animation, as these methods are sometimes called, is relatively easy to implement and often generates satisfactory results at a low processing cost.

Color animation is also used in many other creative manipulations. For example, a sunset scene can be created by increasing the black, red, and orange color saturation of selected screen objects. Or the illusion of movement can be enhanced by having the moving object leave tracks of its image with a decreasing color saturation. This effect, sometimes called a motion blur, is depicted by the bouncing ball shown in Figure 1.10.

### 1.3.4 Object Rendering

In the creation of the image set the animator is often confronted with a modeling problem. As the number of dimensions of the representation and the complexity of the objects increase, so do the difficulties in obtaining the in-between images or the mathematical transformations required for the animation. As a general rule it can be stated that two-dimensional objects are easier to model than three-dimensional ones, symmetrical objects are easier than asymmetrical ones, and geometrical entities are easier than living organisms. However, notice that there are exceptions to these rules.

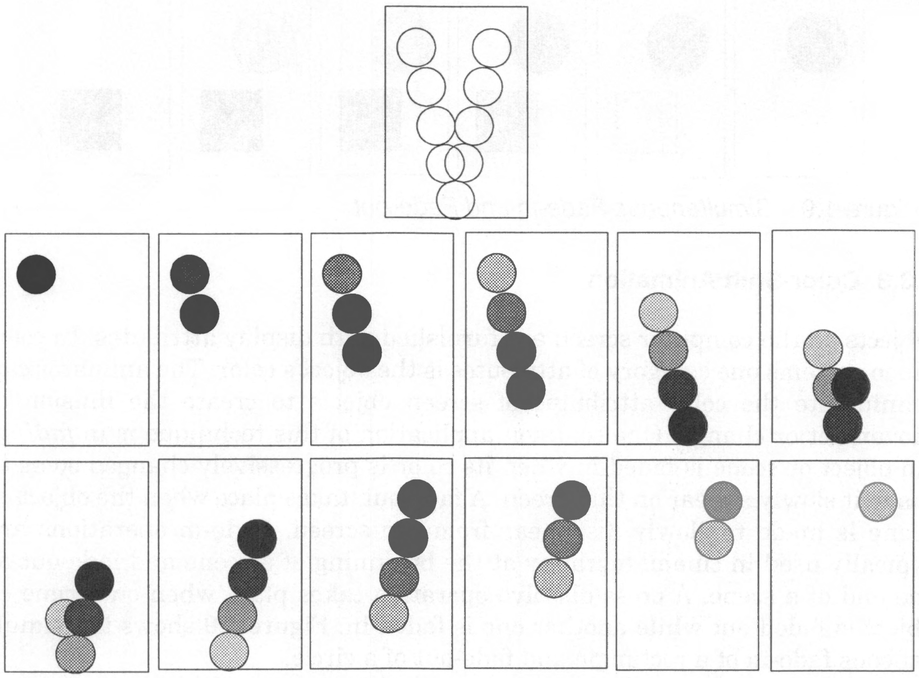


Figure 1.10 *Motion Blur Effect*

Object rendering is closely related to computer animation; as computer rendering methodologies have improved and gained in realism, so has animation. The image set for animating a two-dimensional object can sometimes be obtained by performing geometrical transformations on the object's image (see Figure 1.2). In three-dimensional representations the transformations take on an additional level of complexity. Computer graphics has traditionally resorted to geometrical simplifications that aid in the representation and transformation of three-dimensional objects. One such scheme is based on using polygons to represent objects. For example, a cylinder can be represented as shown in Figure 1.11.

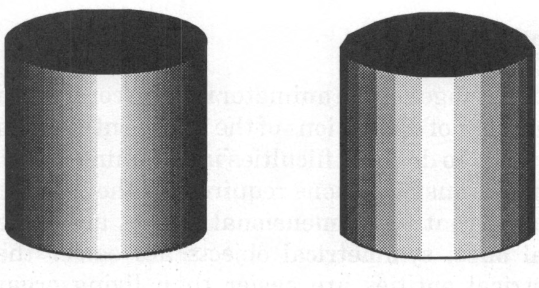


Figure 1.11 *Polygonal Representation of a Cylinder*

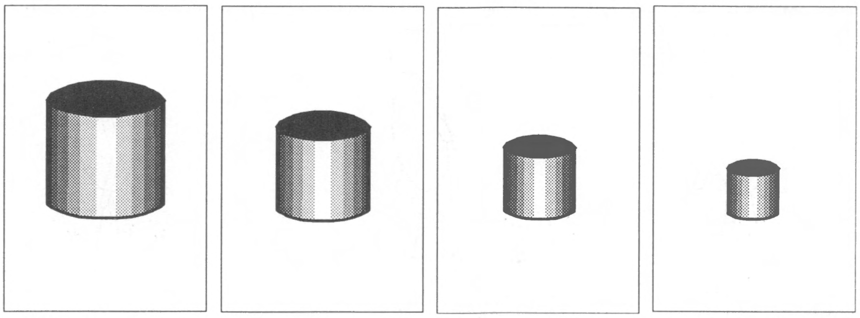


Figure 1.12 *Transformations of a Polygon-Rendered Object*

Although the conventional approach in polygonal representation is to treat each polygon as an independent entity, alternative approaches have been recently developed which offer certain advantages. One such approach is based on representing the polygon edges, rather than the vertices, in the data structures that encode the object's image. This is a simplification over the conventional approach, which requires that the shared edges of adjacent polygons be processed twice.

In either case, the transformation of a polygon-based rendering is a geometrical manipulation of the vertices or edges that define the object. In Figure 1.12 the polygonal representation of the cylinder is easily scaled by proportionally reducing the size of each of the polygons that define it.

The modeling of realistic living organisms introduces difficulties beyond those of geometrical forms. Higher animals and human forms, in particular, present difficult and challenging problems since the conventional mathematical models are not very suitable and muscle action is difficult to predict and imitate. Several techniques have been used to model the human body in three dimensions. Stick figures and surface and volume models have all been used with moderate success. Figure 1.13 is a stick figure of a walking man.

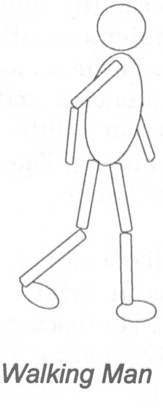


Figure 1.13 *Stick Figure of a Walking Man*

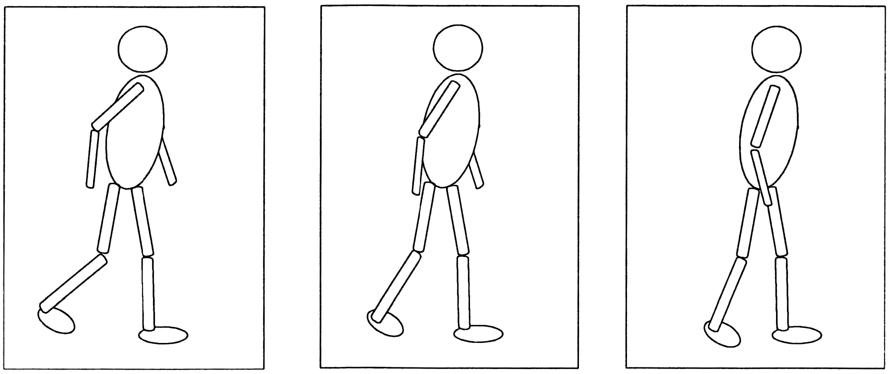


Figure 1.14 *Stick Figure Animation*

The image set required for the animation of a human or animal form cannot usually be obtained by pure mathematical transformations, as is the case with geometrical objects such as the flying dagger in Figure 1.2. Even in the most schematic representations (such as the one in Figure 1.13) the image set involves the interaction of several limbs and joints, as shown in Figure 1.14.

Several techniques have been developed for the computer modeling of human motion. In one methodology (Labanotation) the body is described as sets of limbs and joints. Each joint is specified in terms of axes that can be oriented in various ways. Joint movements are described by operations that fall into several categories. A special symbol represents each class of operation. This approach makes possible the study and representation of human motion in an abstract way.

## 1.4 Applications of Computer Animation

Computer animation is an attribute of the computer graphics environment; to very few fields of this environment can the animation attribute not be applied. Therefore, the applications of computer animation practically coincide with the applications of computer graphics. For instance, computer graphics are often used in business to draw charts of economic and financial functions. The usual purpose of these charts and graphs is to facilitate the understanding of complex phenomena and to aid in decision making. These purposes are enhanced when the graphs and charts are animated so as to represent historical changes or future trends of the depicted data.

In the following sections we describe some fields of computer graphics in which animation plays a central role or in which animation techniques greatly enhance the graphics environment. The discussed applications should be taken as a sampling and not as a restrictive listing.

### 1.4.1 Simulators and Trainers

Many natural or man-made objects and environments can be artificially represented in a satisfactory manner. For many years we have used optical planetariums to illustrate and teach astronomy in an environment that does not require more costly optical instruments and that is independent of the weather and other meteorological conditions. In the planetarium the viewer sits in a comfortable chair, located in an air-conditioned enclosure, and watches the procession of constellations and deep-sky objects, as well as the trajectory of the moon and the planets over a realistic sky. The operator of the planetarium controls the rate of movement so that the celestial transformations that take place over years or centuries can appear to occur in a few minutes. Or the operator can enlarge the magnification of a particular object so that the viewer can appreciate in details the rings of Saturn or the satellites of Jupiter. Furthermore, it is possible in an artificial environment to reproduce the stellar objects and viewing conditions of any particular date in history. In this manner a viewer is able to relive the astronomical observations and experiences of Galileo or Newton.

On the other hand, some natural phenomena cannot be conveniently reproduced in a physical or optical laboratory. For example, the transformation of mass according to the theory of relativity would be practically impossible to reproduce physically. We can use animated graphics to simulate physical entities or to represent complex scientific phenomena such as nuclear and chemical reactions, hydraulic flow, physiological systems and organs, or structures under load; in reproducing physical simulators, such as the planetarium; in depicting systems that cannot be conveniently imitated in other ways; or in creating a more feasible or economical emulation of physical phenomena.

One such type of computer-assisted devices, sometimes called *simulators*, find practical and economical use in experimentation and instruction. Astronauts training for a lunar landing practiced in simulators of the landing module and the mother ship. Airplane pilots often train in computer-assisted simulators that can safely reproduce unusual or dangerous flying conditions.

### 1.4.2 Electronic Games

Since the introduction of Pac Man and similar programs in the mid-1980s computer animation has played an increasingly important role in the personal entertainment field. More recently we have seen the geometrical increase in popularity of dedicated computer-controlled systems and user-interaction devices, such as those developed by Nintendo and Sega. During this time the arcade-type electronic game has continued to prosper.

In the microcomputer world, CD-ROM, digital audio, software, and specialized user-interaction devices have been combined in an environment sometimes called *multimedia*. The quality of the animated imagery and sound effects that can be obtained in multimedia computer systems often competes with those in dedicated systems. Some applications for personal computers have achieved such a degree of realism that moral and ethical issues are being raised regarding the use of sexually explicit or violent applications.

### 1.4.3 Business Presentations and Marketing

In the business environment computer animation often serves to enhance the presentation of graphic and statistical data. In this context the animation can serve to make the presentation more interesting to the spectator by showing transformations that take place over a time period. For example, the evolution of a product from raw materials to its finished form, the growth of a real estate development from a few houses to a small city, or simply the evolution of a statistical trend.

Animated imagery can thus serve to make a more convincing presentation of products or services offered to a client, as a training tool for company personnel, or as a replacement media for presentations of statistical data. As a selling tool computer animation techniques can make a product or service more interesting and also add action and movement to an otherwise dull and boring description of properties and features.

### 1.4.4 Artificial Life

In recent years a new discipline of computer science, named *artificial life*, or *ALife*, has evolved around the computer modeling of biosystems. The new field is said to be based on biology, robotics, and artificial intelligence. The results are digital entities that resemble self-reproducing and self-organizing biological life forms. Computer viruses of the harmful and benign forms are often cited as examples of artificial life.

The cellular automaton is at the core of the notion of artificial life. This idea, first described by John von Neumann, is a theoretical model of a parallel computing device which is subject to various restrictions in order to make possible the formal investigation of its various computing powers. The model is reminiscent of a living organism since it is based on an interconnection of identical cells, each being a finite-state machine. Each unit computes an output based on input received from a finite set of cells, which are said to form its *neighborhood*. It is also possible for a cell to receive input from an external source. A clock tick determines that all cells produce a simultaneous output. The output is directed to all cells in the neighborhood, and possibly, to an external destination or receiver.

The first formal discussion of cellular automata was by E. F. Codd in 1968 (see Bibliography). A. W. Burks is the editor of the book *Essays on Cellular Automata* (1970). A more recent title by Edward Rietman, *Creating Artificial Life: Self Organization,* provides a rigorous, and at the same time, entertaining presentation of this subject.

The implementation of cellular automata is often represented as a sequence of images. Each clock cycle is an iteration update of the automata system, which can be viewed graphically on the computer screen. The resulting changes in the system give rise to an image set that simulates an animated entity. In general, the notion of artificial life is naturally associated with biological forms capable of self-reproduction and self-organization. These actions imply changes that can be represented graphically. In the same way that we associate natural life with movement and action, it often requires the depiction of transformations and movements by means of computer animation.

### 1.4.5 Virtual Reality

Recent breakthroughs in input and output technology have made possible a new level of user interaction with a computing machine, called *virtual reality*. Virtual reality technology consists of a computer system, a viewing device (typically in the form of *virtual reality goggles* or *head-mounted display*), and one or more input devices which allow the user to interact with the animation system.

The result of virtual reality is a digital universe created by the computer system in which the user is more or less immersed, according to its level of isolation from the surrounding environment. This digital universe has been named *cyberspace*, using a term coined by science fiction writer William Gibson in his 1984 book *Neuromancer*. The possible applications of VR technology range from pure entertainment to practical industrial controls. For example, we can put on VR goggles to travel to the planet Mars and walk on its surface or to control a complex robot used in industry or manufacturing. Other possible applications include scientific and medical research, art, music, CAD, electronic games, information management, engineering, education, surgery, and many others.

Animation techniques are usually required in virtual reality as part of the computer feedback mechanism. In a typical VR system the goggles take the place of the video display. The animator uses its art to present to the user a convincing image of the virtual environment created by the system. For example, when the system detects a left-hand movement of the user's head, the video image displayed on the VR goggles is smoothly panned to the left in order to make visible objects that were previously outside of the user's field of view. If the virtual universe includes entities that move, the system must use animation to reflect this action on the virtual environment. For example, a virtual reality representation of the Jurassic period requires that images of dinosaurs move in predetermined or random fashion, perhaps interacting with the user.

Notice that we have not yet achieved the level of technical refinement and the image processing power necessary for creating a realistic virtual environment in which many virtual entities are simultaneously animated according to the user's interaction with the system, or to predefined or random factors. In the years to come we are likely to create virtual realities in which a user is able to experience being a brain surgeon, a time traveler, or a rather skimpy meal for a large, flesh-eating animal of the Jurassic period.

### 1.4.6 Fractal Graphics

When examined closely, natural surfaces are highly irregular and do not follow predictable geometrical patterns. Such is the case with coastlines, islands, rivers, snowflakes, and galaxies. Therefore, most natural objects cannot be satisfactorily represented using polygons or smooth curves, since the resulting image would appear too regular and contrived. However, it is possible to realistically represent some types of natural objects by means of a mathematical entity called a *fractal*.

The term fractal is derived from the words *fractional dimensions.* It is best visualized by means of a structure called a *triadic Koch curve*. The evolution of the Koch curve starts with a straight line of length one. The middle third of this line (one-third fraction) is replaced by two lines of the same length that form a 60-degree angle. The result is a curve that is more rugged than the original one. This second-order curve can be transformed into a curve of the third order by repeating the same process with each of its four segments. The evolution of a Koch curve to the third order is shown in Figure 1.15.

In regards to the Koch curve in Figure 1.15 we observe that its length increases in relation to the number of straight line segments that it contains. This means that the second-order Koch curve in Figure 1.15 has a greater length than the first-order curve. By the same token, the third-order curve has a greater length than the second-order one. Therefore, by continuing the process to infinity, the length of the curve also increases to infinity. In other words, the curve cannot be measured in one dimension. On the other hand, the Koch curve cannot be measured in two dimensions, since, by definition, its area is always zero. This leads to the conclusion that the curve must have a dimension that is greater than one and less than two, that is, a fractional dimension, or fractal. In fact, the dimension of the Koch curve has been determined to be approximately 1.2857 following the Hausdorff-Besicovich method.

The term "fractal" was coined by Benoit Mandelbrot in his book *The Fractal Geometry of Nature* (see Bibliography). One interesting feature of fractals is that they can be generated by computers following what is called a *production rule*. Figure 1.15 shows graphically the production rule for a triadic Koch curve. Other fractals such as the popular Mandelbrot set and the Julia set have their own unique production rules.

The Koch curve exhibits a feature known as *self-similarity*. This means that parts of the curves are similar to the whole curve. Natural objects, on the other hand, rarely exhibit self-similarity, although they do show what is termed *statistical self-similarity*. In using fractal curves to simulate natural objects it is necessary to introduce a random factor that eliminates the curve's self-similarity property. The result is comparable to the image formed in a kaleidoscope, in which the random placement of the colored glass fragments ensures a unique image with every change.

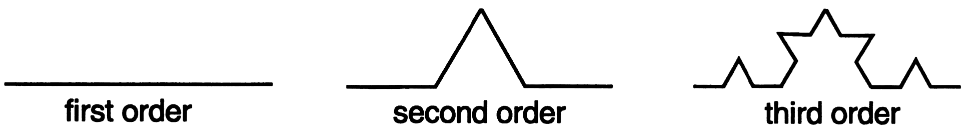


Figure 1.15 *Triadic Koch Curve*

Computer animation can be used to show the progression in the approximation of random fractals in a computer system. Notice that a truly random fractal has an infinitely complex shape; therefore it cannot actually exist as a visible object. The introduction of a random element in the creation of the fractal curve ensures that the result will be unpredictably different every time that the fractal is approximated. The animated imagery that results from the generation of a random fractal graphic approximation is quite interesting from both an artistic and a mathematical viewpoint.

## 1.5 The Animator's Predicament

The PC computer animator working with present day technology will rarely have sufficient resources for the purpose at hand. A typical scenario consists of a short supply of one or more of the necessary elements required for image processing or rendition. For example: the CPU or the coprocessor do not have the processing power to perform the necessary image transformations, and the video image definition and color range do not allow the satisfactory representation of real world objects or beings.

For these reasons, the result of an animation effort in a small computer environment can very easily result in a bumpy, coarse, and unrealistic imagery that is aesthetically unpleasant and even physiologically disturbing. The animator's art consists of making the best possible use of limited resources in processing and image representation in order to produce a result that is as smooth and pleasant as the media allow. This often requires stretching the system's capabilities to its extremes as well as resorting to every scheme and stratagem in a programmer's bag of tricks.

Most of the programmer-animator's work consists of making compromises and in finding acceptable levels of undesirable effects. In this sense the animator often has to decide how small an image satisfactorily depicts the object, how much bumpiness is acceptable in representing a movement, how little definition is sufficient for a certain scenery, or with how few colors can an object be realistically depicted. In the hands of the expert, these compromises and concessions result in the best possible representation in a particular system.

The expert viewer, who is familiar with the hardware limitations of the media, often appreciates and even marvels at the animator's achievements. The nontechnically oriented user, on the other hand, usually compares the result with those possible with other animation vehicles, and points out that the computer images are not as good as television or as cinematography. This means that a computer-animated program for the PC environment, so well designed and executed that it manages to escape these harsh comparisons and judgments of the typical user, is indeed a technological and artistic accomplishment.

Chapter

# 2

# Graphical Image Structures

## 2.0 Image Storage for Animation

The typical scenario is that animated action takes place within the structure of a graphics program. The animated sequence is usually a set of graphics images displayed on the computer's video system. Certain image structures and encodings are more animation-friendly than others. Therefore, before we tackle the problems of the animated image set, we must first consider how the image is encoded and stored. For these reason, the selection and layout of the data structures that contain the image data are one of the most important considerations in the design of an animated application.

### 2.0.1 Pixel Maps versus Vector Commands

In general terms, computer images can be classified into two categories: pixel maps and vector commands. A pixel map, or bitmap, is a memory structure that encodes the relative location and the attribute of each light dot (pixel) that forms the image. Alternatively, the graphics image can be defined by means of vector or display file commands for each of the image's geometrical elements. Figure 2.1 shows the image of a cross defined as a bitmap and as a set of vector commands.

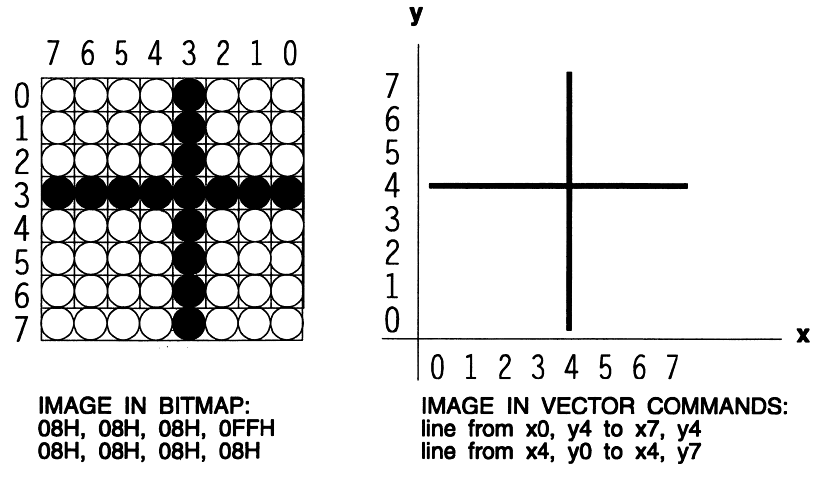


Figure 2.1 *Image Encoded in Bitmap and Vector Commands*

In Figure 2.1 the bitmap represents the attribute of each individual pixel in the image. In the simplest encoding a 0-bit in the bitmap usually represents a white or uncolored pixel and a 1-bit a black or colored pixel. Vector commands refer to the geometrical elements in the image. For example, the vector commands in Figure 2.1 define the image in terms of two intersecting straight lines. The commands contain the start and end points of each line in a cartesian coordinate plane that corresponds with the system's video display.

The question of which of these two methods of image encoding is preferable has no unequivocal answer. The most suitable approach for many applications is to adopt both methods of image encoding. Which is preferred depends on occasional image characteristics and processing requirements. A video image composed exclusively of geometrical elements, such as a line drawing of a building or a machine part, can usually be defined flexibly and compactly by means of vector commands. On the other hand, a naturalistic representation of a human face usually requires a bitmap.

Each method of image encoding, bitmap and vector commands, has its own features and advantages. One consideration is that vector commands sometimes save considerable storage space over bitmaps. For example, in a video surface of 600-by-400 screen dots, the bitmap for representing two intersecting straight lines would have to encode the individual states of 240,000 pixels. If the encoding is in a two-attribute form, as in Figure 2.1, then one memory byte is required for each 8 screen pixels. The result is that a 30,000-byte memory area is devoted to storing the bit-mapped image. On the other hand, the same image could be encoded in two vector commands that define the start and end points of each line, with a considerable saving in storage. By the same token, to describe in vector commands a screen image of Leonardo's painting of the Mona Lisa would certainly be more complicated and memory consuming than the corresponding bitmap.

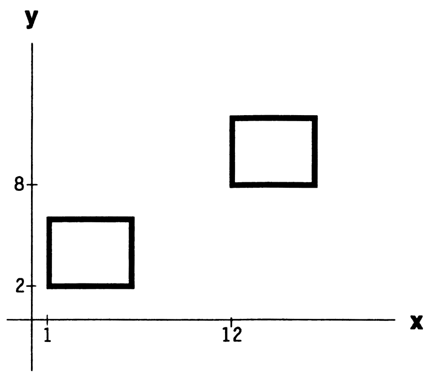


Figure 2.2 *Translation by Coordinate Arithmetic*

A second consideration regarding bitmaps versus vector commands is that vector commands locate geometrical image elements by means of coordinate points. Graphics software can operate mathematically on these points to transform the encoded images. For example, a geometrically defined object can be moved to another screen location by adding a constant to each of its coordinate points. In Figure 2.2 the rectangle with its lower left-most vertex at coordinates $x = 1, y = 2$, is translated to the position $x = 12, y = 8$, by adding 11 units to its $x$ coordinate and 6 units to its $y$ coordinate.

## 2.1 Device-Independent Graphics

Graphics software systems and applications can often be envisioned and designed independently of any particular graphic device. Nevertheless, the ultimate purpose of a graphic system is to create a picture on a physical instrument. Therefore, computer graphics cannot exist separately from a computer graphics device. This means that the notion of device-independent graphics makes more sense as a design goal that as a programming reality.

Graphics software services are typically furnished in the form of a graphics library, a graphics standard, or a graphics programming language. Several graphics standards and languages have been developed, mainly for the purpose of providing some degree of device independence to the software medium. Additionally, some operating systems, such as Windows and OS/2, have taken on the task of providing device independence to applications.

In the MS-DOS environment device independence is the objective of the program designer aiming at a software product capable of executing in more than one graphical input or output device. Suppose a graphics library that contains two services, one for drawing straight lines and one for drawing circles. Assume that this hypothetical library is furnished with two drivers, one for VGA and another one for the XGA video system. Also assume that the parameters for the line-drawing service are the cartesian coordinates of the line's end points and that the parameters for the circle-drawing service are the coordinates of the center of the circle and its radius.

The problems of device independence become immediately evident, even in the simplest conceivable application. In the first place we must take into account the hardware differences between the VGA and the XGA systems. For example, the maximum vertical definition of the VGA is 480 pixel rows, while the XGA is capable of displaying 768 rows. Therefore, if 500 is entered as a pixel row value, it would be valid if the system were XGA, but not so if it were VGA. The possible conflict must be addressed by the device-independence engine.

There are several possible approaches to the problem of executing in dissimilar devices. The easiest to implement, but perhaps the least satisfactory solution, is to limit the resolution to that of the least powerful device. In this example, device independence could be ensured by limiting the resolution to that of the VGA system, sacrificing the XGA modes that exceed the VGA definition. The approach ensures uniformity by reducing the system's graphic potential to that of its lowest component.

Another option is to provide compensations in the core computational routines in order to accommodate the characteristics of different hardware. One disadvantage of this approach is that the software package must take into account the operational characteristics of all supported devices. Therefore, the entire system would have to be modified in order to extend support to a new device with different hardware characteristics and display parameters.

A third approach to device independence is to perform the necessary compensations and adjustments, not in the core computational routines, but in separate software units configured according to the characteristics of each supported device. These hardware-specific units are usually called *device drivers*. The central software package can be based on an imaginary model of a virtual graphics system, for example, on a screen structure of 1600-by-1000 pixels. The line-drawing and circle-drawing routines compute pixel position for this virtual video display. The chore of adjusting the imaginary pattern of screen dots to the parameters of the physical device is left to each device driver.

There is no ideal solution to the problems created by the use of dissimilar, and sometimes incompatible, hardware devices. In alphanumeric modes, or in the case of undemanding graphics applications, some degree of device independence can be achieved, as is evidenced by the Windows and OS/2 operating systems. As code attempts to make optimum use of the hardware features, device independence becomes, progressively, a goal more difficult to achieve, as is seen in the following sections.

### 2.1.1 Software Environment for PC Animation

In the PC environment, animated programs that achieve satisfactory results often do so by pushing the graphics hardware to its processing limits. To the animation programmer every processing operation is critical and every system capability is in short supply, because each microsecond of execution time is crucial to producing a satisfactory result. Performance concessions and processing complications which are required to ensure device independence detract from the hardware efficiency and therefore diminish the quality of the animation.

For these reasons, to the present day, the most satisfactory animated applications for the PC take control of the graphics hardware in order to exploit the capabilities of a particular device to its maximum potential. This means that animated programs typically execute under MS DOS, as a DOS application in Windows or OS/2, or as a Windows or OS/2 application with I/O privileges. Most of the animation programming techniques presented assume that the software has total hardware control, as is the case in the above environments. At the time of this writing, a satisfactory quality in computer animation cannot yet be achieved by means of operating system or other high-level services.

## 2.2 A Virtual Graphics Machine

The high degree of hardware control required in animation does not mean that the program designer must completely renounce the notion of device independence. The fact that an animated application makes optimum use of the hardware facilities implies that the program must be equipped with device-specific routines. But there is no limit to how many different devices are supported by a particular program. Here again, the program designer determines the hardware systems supported as well as the support approach. The designer's strategy can go from developing a separate program for each supported device (which would ensure the best possible execution) to implementing a single program version that executes, more or less acceptably, in all the supported systems. In any case, by following specific methodologies the program designer can simplify the conversion problems and increase the portability of the code so that support of multiple systems is made as uncomplicated as possible.

One approach is to adopt an imaginary model of a graphics device. This model is called a *virtual graphics machine*. The characteristics of the virtual machine occasionally coincide with those of a physical device. More frequently, the virtual machine has characteristics that exceed those of the most powerful physical device available. In this manner the program designers attempt to leave room for future improvements in the graphics hardware.

Conceptually, the notion of a virtual graphics machine includes that of a graphics engine. If the physical graphics device is the hardware counterpart of the graphics machine, the graphics engine is the functions which the virtual device is capable of executing. Therefore, the specifications of the graphics machine include the following elements:

1. The hardware characteristics of the adopted model, called the virtual graphics device or VGD
2. The graphics functions that can be directly performed by the device, called the output functions or graphics primitives
3. The user interaction with the device, called the input functions

4. A structured filing system adopted for storing, restoring, and manipulating the graphic image, called the display file

### 2.2.1 The Virtual Graphics Device

The VGD is an abstract model, although, for practical reasons, some designers make it coincide with one of the hardware devices supported by the system. In this case, the virtual graphics device matches a physical device.

The VGD is usually defined during the program design stage. At this time it is important to make reasonable assumptions regarding the characteristics and capabilities of the devices that are supported by the software. If the model adopted substantially exceeds the capabilities of the best physical devices available, the system is unnecessarily elaborate and complex. In this case it can be described as being overdesigned or overspecified. By the same token, if the adopted model has fewer capabilities than the physical devices available, some of the graphics power of the hardware is lost to the software and the system. In this case the system can be said to be underdesigned or underspecified.

In conventional computer graphics the VGD is usually an imaginary display system. The surface of this display is viewed as a two-dimensional cartesian coordinate system. In graphics programming it is convenient to place the origin of the coordinate system in the top left-hand corner, because the rows and columns of pixel-based displays are usually referenced from this position. Since in conventional cartesian notation this quadrant contains negative values for $y$ coordinates and positive values for $x$ coordinates, an adjustment is made in the convention so that $x$ and $y$ are both positive.

The maximum $x$ coordinate is the horizontal definition and the maximum $y$ coordinate is the vertical definition. Figure 2.3 shows a virtual graphics device in the form of a video display with a definition of 900-by-1600 pixels.

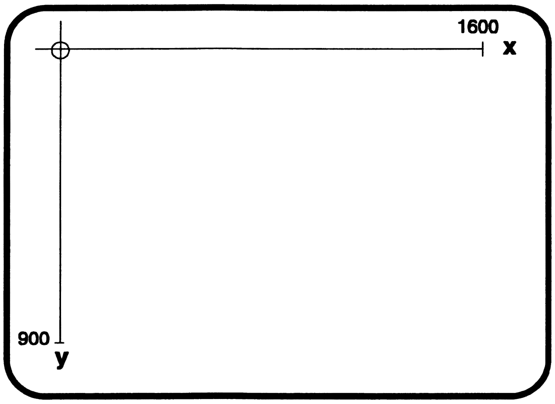


Figure 2.3 *Cartesian Representation of the Video System*

In addition to the coordinate range and definition, the program designer must also determine other capabilities of the virtual graphics device, such as the number of colors supported by the system. It is important to differentiate between the colors that can be selected and those that can be displayed. For instance, a graphics system may be capable of displaying 16 colors simultaneously that can be selected from a total of 128 available hues and shades. The available colors are sometimes called the palette. The number of simultaneous colors is the system's color range.

The horizontal and vertical definition, the color range, and the color palette are sufficient to describe the functional characteristics of the virtual graphics device adopted as a model for a given project. For example, the design specifications document could state that the display device has a definition of 640-by-480 pixels in 16 colors that can be selected from a palette of 64 colors.

### 2.2.2 The Graphics Primitives

A graphics system is an imaging tool; therefore it must be capable of performing elementary graphics functions, such as drawing lines and geometric figures, displaying text characters, and shading or coloring screen areas. The available image-creating operations are called the output functions or graphics primitives of the system.

A general purpose graphics library generally includes a more or less extensive collection of graphics primitives. An application, on the other hand, includes only those functions required for its specific purpose. A minimal, general purpose graphics library contains the following primitives:

1. Full screen primitives: clear the screen, set the entire screen to a color or attribute, save the screen image in memory, and restore a saved screen image.
2. Screen tile (window) primitives: set a rectangular screen area to a given color or attribute, save a rectangular screen area in memory, and restore a saved rectangular screen area.
3. Attribute selection primitives: set the current drawing color, set the current fill color, set the current shading attribute, set the current text color, set the current text font, set the current line type (continuous, dotted, dashed, etc.), and set the current drawing thickness.
4. Geometrical primitives: draw a straight line, draw a circular arc, draw an elliptical arc, draw a parabolic arc, draw a hyperbolic arc, and draw Bezier curves.
5. Image transformation primitives: scale, rotate, translate, and clip image.
6. Painting primitives: fill a closed figure with current fill color or shading attribute.
7. Bit block primitives: XOR text or bit block, AND text or bit block, and OR text or bit block.

### 2.2.3 Input Functions

The computer graphics system must usually be capable of interacting with a human element. This takes place through an input device such as a keyboard, a mouse, or a graphical input tablet. This input can be roughly classified into two types: valuator and locator.

Valuator input takes place when the data entered is an alphanumerical value. For example, the coordinates of the end points of a line constitute valuator input. Locator input takes place when the user interaction serves to establish the position of a graphic object called the locator. A mouse-controlled icon is a common locator.

Valuator and locator input normally follow this sequence of input phases:

1. Input request phase: The graphics system goes into the input mode and prompts the user that it awaits further action.
2. Echo phase: As the user interacts with the input device, its actions are echoed by the graphics system. For instance, the characters are displayed as they are typed, or the icon moves on the screen as the mouse is dragged on its surface. Phases 1 and 2 are sometimes called the prompt-and-echo phase.
3. Trigger phase: The user signals the completion of input by pressing a specially designated key or a button on the input device. One way to conclude the input phase is to abort the operation, usually by pressing the escape or break key.
4. Acknowledge phase: The graphics system acknowledges that the interaction has concluded by disabling the input prompt and by notifying the user of the result of the input. In the case of locator input the acknowledge phase often consists of displaying a specific symbol that fixes the locator position. In the case of valuator input the acknowledge phase can make the cursor disappear. Another action of the acknowledge phase can be that the characters entered are reformatted and redisplayed, or they are stored internally and erased from the CRT.

A general-purpose graphics library includes the following interaction primitives:

1. Valuator input primitives: input coordinates, input integer, input string, and input real number.
2. Locator selection primitives: select cursor type (crosshair, flashing rectangle, rubber band, or others).
3. Locator input primitives: enable and disable screen icon, move screen icon, and select graphics item on screen and menu item.

### 2.2.4 Display File Structure

A graphics application must be capable of storing and transforming graphics data. The logical structure that contains this data is called the *display file*. One of the advantages of a display file is that it allows the compact storage of graphics data and its transformation through logical and mathematical operations. For example, an image may be enlarged by means of a mathematical transformation of its coordinate points, called a scaling transformation. Or the graphics object can be viewed from a different angle by means of a rotation transformation. Another transformation, called translation, allows changing the position of a specific object.

Before these manipulations can take place, the program designers must devise the logical structure that encodes image data in a form that is convenient for the mathematical operations to be performed. High-level graphics environments, graphical languages, and operating systems with graphics functions provide precanned display file structures that are available to applications. The programmer working in a customized environment, on the other hand, usually designs the display file to best accommodate and manipulate the data at hand. The first step in defining this structure usually consists of standardizing the screen coordinates. Figure 2.3 shows the normalized screen coordinates of a virtual video display system.

A screen normalization scheme usually aims at maximum simplification. One possible scheme is to select the top-left corner of the screen as the origin of the coordinate system and make all locations positive (see Figure 2.3). The range of values that can be represented in either axis determines the system's definition. If an application is to support a single display definition, it is convenient to normalize the screen coordinates to this range. However, this decision should be taken cautiously, since equating the virtual to the physical device means that any future support for a system with a different definition probably implies modifying the entire software package.

Notice that screen normalization is necessary so that image data in the display file can be shown on a physical device, but the stored image data does not have to conform with the adopted screen normalization. At display time the processing routines (usually in the device driver) perform the image-to-pixel conversions. In Chapter 3 we describe the operations necessary for converting data in the image file into displayed pixels on the video screen.

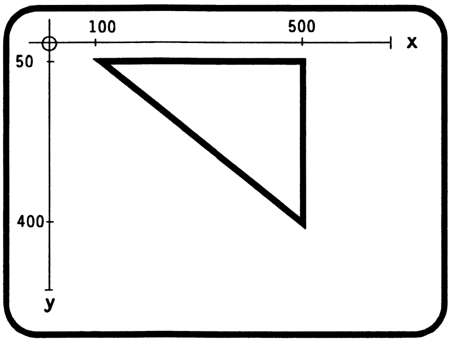
### 2.2.5 Image Data in the Display File

How the image is stored in the display file depends on the image itself and on the operations to be performed on its elements. Graphical images are classified into geometrical and bit-mapped; therefore, with every image to be stored, a decision must be made whether to represent it as a set of vector commands, as a bitmap, or as a combination of both. In many cases the image itself determines this decision. For example, there is little doubt that a circle is best encoded in vector form. On the other hand, images such as alphanumeric characters can be represented either as vector commands or as bitmaps. Postscript and other conventions have used vector representation of text characters in order to facilitate scaling.

Even after deciding if a graphics object is to be represented as a bitmap, as a set of vector commands, or as both, there can be considerable variation in the encoding. A straight line can be defined by its two end-point coordinates, or by its start point, angle, and length. A rectangle can be defined by the coordinates of its four vertices, or by the coordinates of two diagonally opposite vertices. The first option allows the representation of parallelograms, while the second one is more compact. There are also variations in the encoding of bit-mapped objects. If the object is unique, its bitmap can be included in the display file. However, if the application is to manipulate several objects with the same bitmap, then it is better to represent the bitmap with a special type code in the display file and store a single, generic bitmap in a separate location. The design of the image data formats for a customized display file requires careful consideration and planning. Even then, it can usually be anticipated that as the program is developed, changes in the image data encoding become necessary to accommodate or facilitate operations, or to compress the information. The safest approach in the development stage is to test all processing operations with minimal image data, instead of proceeding to encode elaborate images into data formats that may later require modifications.

### 2.2.6 Display File Commands

It is not sufficient for the graphics system to store image data. It must also be capable of manipulating this data in order to generate and transform images. The orders that operate on image data are the display file commands. The image itself is defined in terms of both data and commands. For example, a screen triangle could be represented by three straight lines. The display file contains the coordinate points of the three lines as well as the commands to draw these lines, as shown in Figure 2.4.



| DISPLAY FILE | | | | |
|---|---|---|---|---|
| commands: | image data: x | y | x' | y' |
| line | 100 | 50 | 500 | 50 |
| line | 500 | 50 | 500 | 400 |
| line | 500 | 400 | 100 | 50 |

Figure 2.4 *Display File for a Triangle*

Notice that in Figure 2.4 the screen coordinates coincide with the display file coordinates. This simplification, although convenient, is not always the preferred approach.

## 2.3 Graphics Software Standards

Since the late 1970s several organizations have labored towards an international standard for computer graphics. In the United States the Graphics Standard Planning Committee of the American National Standards Institute (ANSI) has developed the *Core Graphics System*. The German standards group created a computer graphics standard known as the *Graphical Kernel System*, or GKS. In October 1981, GKS was submitted and approved by the International Standards Organization (ISO) as a proposed standard for computer graphics. Since then, GKS has gone through several testing and modification stages. In 1985 it reached its present status of an international standard.

In addition to GKS, other computer graphics standards are under development by the American National Standards Institute. Among them are the *Virtual Device Interface* standard (VDI) and the *Virtual Device Metafile* standard (VDM).

For reasons of performance animation programming must often be done outside of standards and other formal conventions. For this reason, graphics standards such as GKS, although a core topic of general graphics programming, are not discussed in this book.

### 2.3.1 Graphics Support from System Software

In an effort to expedite and standardize graphic programming from high-level languages, IBM and other companies have developed several system-level graphics software products. One of these packages, named the IBM Professional Graphics Series, was intended for the original video systems of the PC line, namely, the Color Graphics Adapter, the Enhanced Graphics Adapter, and the PCjr. This package included an implementation of the Graphical Kernel System, a Virtual Device Interface, a file system manager, and a terminal emulation program. IBM also made available a Graphics Development Toolkit version 1.2 for DOS systems. This package included eight Virtual Device Interface device drivers, five of which are for PS/2 displays, as well as a printer, plotter, and mouse drivers. The IBM Operating System/2 Graphics Development Toolkit is a similar package intended for OS/2 multitasking applications.

In addition, operating system programs provide graphics services that can be employed by high-level and low-level languages alike. Windows and OS/2 programmers have available these graphics functions, which include some limited animation commands. MS DOS, on the other hand, does not contain graphics services, although several versions of IBM Basic Input/Output System (BIOS) provide graphics services which can be accessed by DOS programs.

The BIOS graphics services are included as part of the video functions of interrupt 10H. They afford a software mechanism for reading and writing individual pixels and for setting the graphics mode, for displaying text on a graphics screen, and for manipulating the palette registers. Although the BIOS graphics services are insufficient for completely implementing a graphics application, they do assist the programmer in performing noncritical functions. The use of the BIOS services in graphics programming is discussed later.

## 2.4 Storage of the Graphical Image

The raster-scan video display technology used in the PC interprets a graphical image as a two-dimensional arrangement of light cells, called pixels. In some systems, these light cells are illuminated in monochrome light, while other systems use light of various colors or intensities. The literal storage of a graphical image (bitmaps) requires one discrete storage unit for representing the attribute of each screen pixel.

Graphics images can be represented and stored as a set of vector commands. This geometrical encoding has several advantages, such as a more compact representation, as well as certain possibilities of transforming the stored image by manipulating its coordinate data.

### 2.4.1 Geometrical Image Elements

An elaborate graphics image can often be geometrically encoded by subdividing it into component elements. Not all graphics systems, languages, or applications use the same number or category of image elements, nor are these elements identically defined or named. Nevertheless, there are some fundamental concepts of image encoding that transcend specific implementations.

#### The Point

The primary geometrical image element is a point. In raster-scan technology, such as the PC video systems, it is tempting to equate a geometrical point with an individual screen pixel. But it is more consistent with the principles of device independence to use the concept of a screen point in reference to the virtual display surface, and to reserve the word pixel for the screen element. Figure 2.5 shows the cartesian representation of a point in the first quadrant.

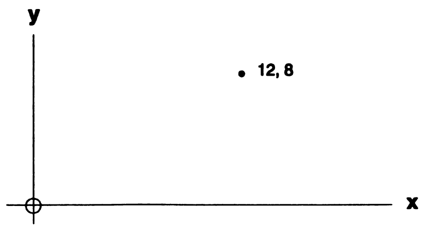


Figure 2.5 *Cartesian Representation of a Point*

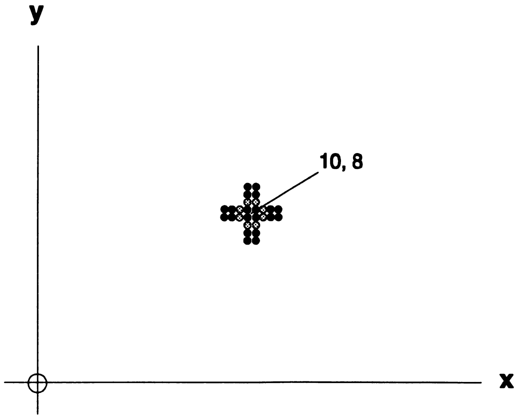


Figure 2.6 *A Screen Point Used to Locate a Bitmap*

Geometrically, we can define a screen point by its *x* and *y* cartesian coordinates. By convention, the first variable in the pair is the horizontal coordinate *x*, and the second one is the vertical coordinate *y*.

In graphics programming the concept of a screen point can be extended to locate a more complex image. In this manner a screen point can reference the placement of a bitmap or other screen object. In Figure 2.6 the coordinates $x = 10$, $y = 8$ refer to the center point of the bitmap image.

## The Line

A line segment can be intuitively defined as those points along a straight line that lie between two end points. This concept is valid geometrically as well as graphically because a straight line in the cartesian plane can always be specified by the coordinates of its two end points. Figure 2.7 shows such a line.

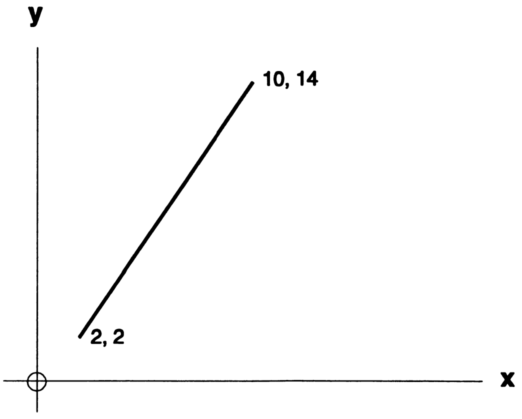


Figure 2.7 *Cartesian Representation of a Line*

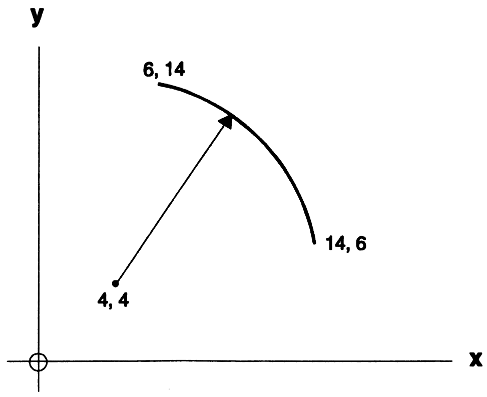


Figure 2.8 *Cartesian Representation of a Circular Arc*

## Curves and Arcs

A curve can be defined as a set of points forming a continuous line and an arc as a part of a curve. This definition does not exclude the possibility of an arc consisting of the entire curve. In this sense a circle is considered an arc. Nor is the concept of curve limited to any type or group of curves, except that for practical plotting purposes the curve must be expressible in a mathematical formula. Figure 2.8 shows a circular arc on the cartesian plane.

In generalizing the representation of curve we see that its encoding requires the following elements:

1. The coordinates of the start point and end point of the arc.
2. A mathematical or literal description of the curve of which the arc is part. This description can be in the form of a verbal expression, a code, or a mathematical formula.
3. The drawing direction if more than one arc can be generated between the end points of the curve described. For example, in Figure 2.9 we see that from given start and end points two arcs can be generated, one in the clockwise direction and another one in the counterclockwise direction.

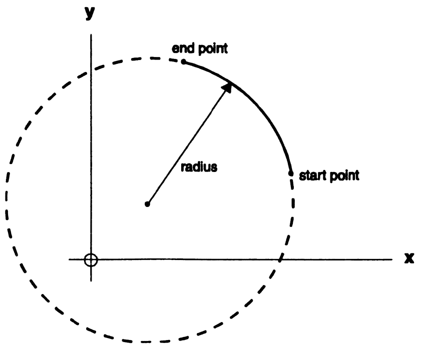


Figure 2.9 *Ambiguity in Encoding a Circular Arc*

4. The necessary data to define the particular curve in the cartesian plane. For example, in the case of a circle, the specification includes the radius; in the case of an ellipse it includes the major and minor semi-axes.

## Polygons

A polygon is defined as a surface bounded by line segments. There is no limit to the number of line segments contained; since the polygon is a closed figure, the start point and the end point in a polygon must coincide. In a polygon, the line segments are also called edges, and the coordinates of the start and end points of these segments are the vertices.

The outline of the polygon can be specified by a series of line segments. Since, by definition, the polygon is a surface, the figure is usually defined in terms of number of sides and the coordinates of each of the vertices.

To simplify the identification of special types of polygons, other data may be optionally included in the specification. For example, in a regular polygon all the line segments are of equal length and support equal angles. Some polygons, such as the triangle, rectangle, square, and pentagon, are so frequently used that they are usually defined independently.

Polygons can also be classified as convex and concave. In a convex polygon, a line segment connecting any two points inside the polygon lies entirely inside the polygon. Figure 2.10 shows convex and concave polygons. The dashed line is used to show the convexity rule.

Often a graphics application must fill the surface of a polygon with a given pattern or color. The concavity or convexity condition must be taken into account during the polygon fill operation.

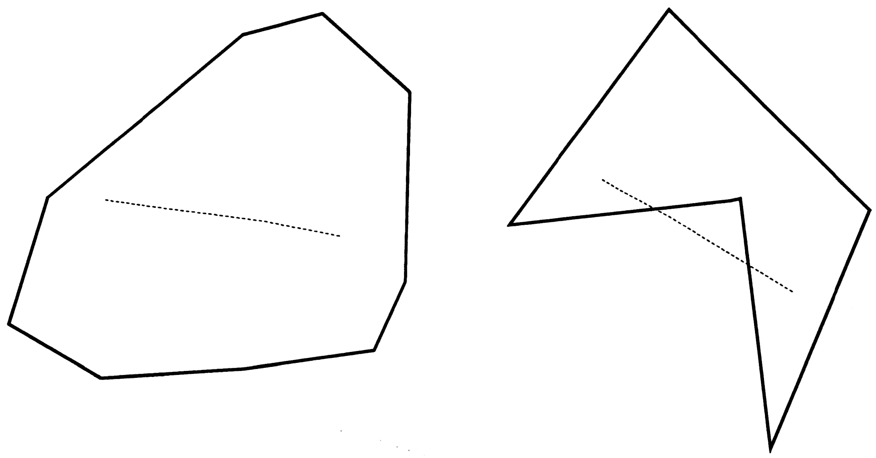


Figure 2.10 *Convex and Concave Polygons*

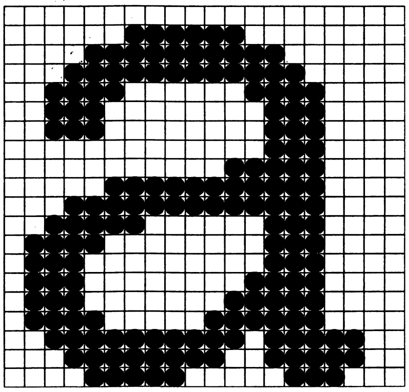


Figure 2.11 *Bitmap of the Letter "a"*

### 2.4.2 Nongeometrical Image Elements

A bitmap (sometimes called a cell) is a nongeometrical image element usually defined as a stored array of points. All elements in a bitmap can share a common attribute, or each element can have its individual attribute. Bitmaps are used to represent an object that cannot be conveniently defined geometrically. Figure 2.11 is a bitmap of the lowercase letter "a."

A bitmap is usually defined by its horizontal and vertical dimensions and by the memory address of the first item in the array.

## 2.5 Image Mapping

The graphical image exists in the physical universe. The typical medium is either a binary storage device or a pixel-mapped display surface. In both cases there are certain concepts, terminology, and logical structures that find frequent use in image mapping, storage, and retrieval.

### 2.5.1 Video Buffer

The video buffer is the portion of physical memory reserved by the system for storing the video image. It is a system-specific concept: the location and structure of the video buffer depends on the architecture of the specific graphics hardware and software. In MS-DOS video systems the video buffer architecture changes in the different display modes. For example, in VGA mode 18 the video buffer consists of four color planes, each plane storing a 640-by-480 pixel image, while in mode 19 the video buffer consists of 320-by-200 pixels, each of which is mapped to a memory byte that encodes the pixel's attribute. The video buffer is also called the display buffer, the regen buffer, the video memory, and the video display buffer. The term frame buffer is also used occasionally and somewhat imprecisely.

Most display systems used in the PC allow access to the video buffer by the CPU (programmer's port) and by the display hardware (video controller's port). For this reason it is often described as a dual-ported system.

### 2.5.2 Image Buffer

While the video buffer is a physical entity, the notion of an image buffer is a logical one. In Section 2.2 we referred to a virtual graphics machine or device consisting of an imaginary display with fictitious characteristics. The concept of the image buffer is usually associated with the virtual graphics device. Since the attributes of the virtual machine can exceed those of the physical one, the dimensions and attribute range of the image buffer can exceed those of the video buffer.

Imagine a PC video graphics system equipped with a video buffer suitable for holding an image of 640-by-480 pixels. In such a system it is feasible to envision a graphics application that supports an image buffer capable of storing 1280-by-960 pixels. In this case, the image buffer quadruples the storage capacity of the video system. Figure 2.12 depicts the case of an image buffer that is larger that the video buffer. In this example the image information stored in the image buffer contains several times the amount of data that can be displayed as a single screen image.

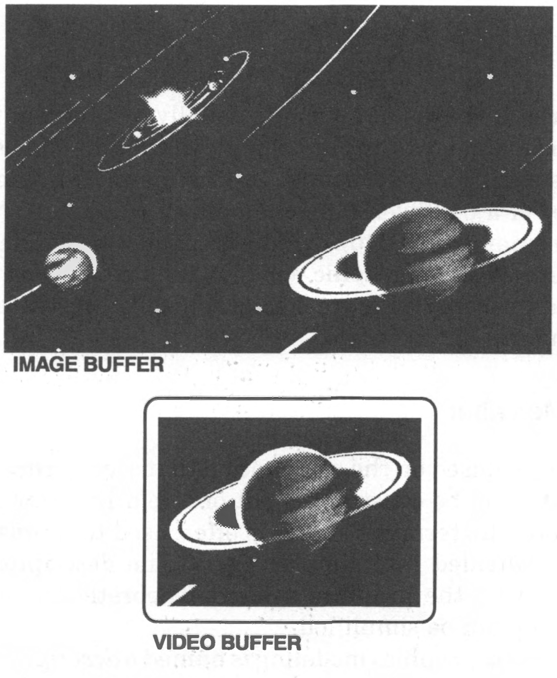


Figure 2.12 *Image and Video Buffers*

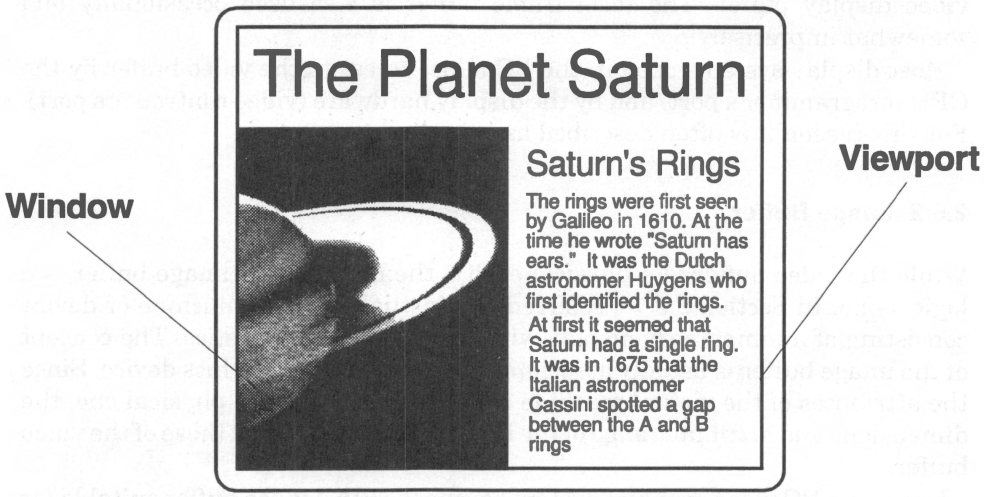

The Planet Saturn
Saturn's Rings
The rings were first seen by Galileo in 1610. At the time he wrote "Saturn has ears." It was the Dutch astronomer Huygens who first identified the rings. At first it seemed that Saturn had a single ring. It was in 1675 that the Italian astronomer Cassini spotted a gap between the A and B rings
Window
Viewport

Figure 2.13 *Viewport and Window*

### 2.5.3 Viewport

The viewport is the area of the display used for graphic operations. Because the PC graphics modes always use the entire display surface, the graphic viewport always coincides with the physical display.

### 2.5.4 Window

A window is an area of the display surface, usually rectangular in shape, which is defined in any convenient way. For example, a window can be defined by the coordinates of its start and end points. In this manner, a window filling the upper-left quarter of a 640-by-480 pixel display would have start coordinates (0,0) and end coordinates (320,240). Windows can also be defined descriptively; for example, we speak of the graphic window, the text window, and the menu window of a certain viewport. Figure 2.13 graphically illustrates the notions of viewport and window.

### 2.5.5 Graphics Modeling

Graphics modeling is based on the assumption that any picture, no matter how elaborate or ornate, can be constructed out of a relatively few, simple components. In a drawing, the term *descriptor* is often used to represent an element that cannot be subdivided into simpler parts. The descriptor concept is an abstraction adopted by the graphics system. Theoretically, any geometrical figure except a point can be simplified.

The second element of graphics modeling is named a *description*. A description is defined as a sequence of at least one descriptor. A graphics model is the representation of objects using literal or mathematical descriptions. In functional and object-oriented programming languages, the model is a representation of the object, but not an instruction to display it. The format and syntax of the model and the available descriptors vary with each language and implementation.

For example, a logical structure for a simple graphics modeling system could be based on the following descriptors:

***move (x,y)*** is a command to set the current location at coordinates *(x,y)*

***line (x,y)*** is a command to draw a line from the current location to a location with coordinates *(x,y)*

***circle (r)*** is a command to draw a circle of radius *r* with its center located at the current location

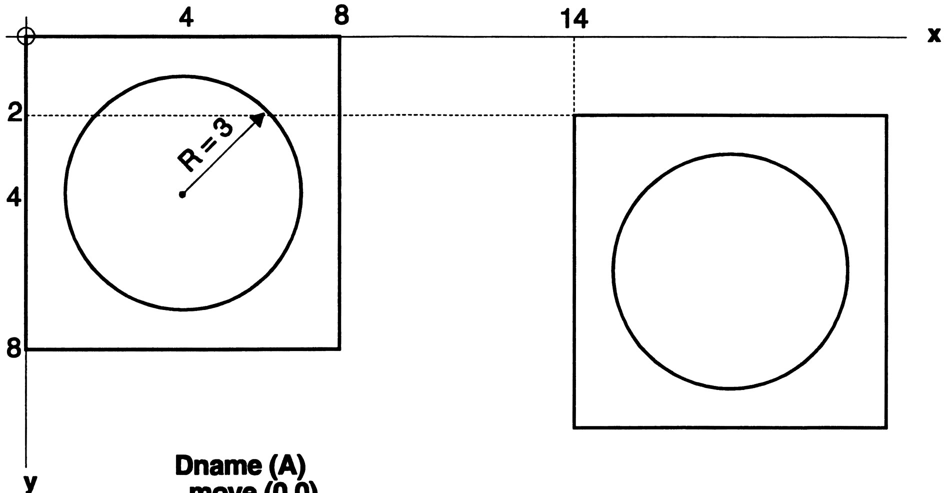
A description can include as many descriptors as necessary in order to represent the figure. In some languages, descriptions can be assigned a variable name. The following description encodes the operations necessary to draw a circle enclosed by a square:

```
Dname (A)
 move (0,0)
 line (8,0) > line (8,8) > line (0,8) > line (0,0)
 move (4,4)
 circle (3)
A ends
```

Notice that the operator *Dname* is used in this example to mark the start of a description and the operator *ends* is used to signal its end. Also note that the greater-than symbol (>) is used to separate descriptors in the same line, as well as to indicate program flow. The above symbols and structures have been invented by the authors for the purpose of the current illustration, and they do not correspond with the actual operators of any known graphics language or system.

The model of a graphic object may also specify transformations to be performed on its description. These transformations are the usual operations of translation, rotation, scaling, and others previously mentioned. In some languages, the transformed description is called a graphical object. A possible scheme for representing transformations in a graphical language can use parenthesis, brackets, and capital letters, as in the following example of a translation of the graphical description A:

```
SHIFT (14,2) [A]
```



```
Dname (A)
  move (0,0)
  line (8,0) > line (8,8) > line (0,8) > line (0,0)
  move (4,4)
  circle (3)
A ends
SHIFT (14,2) [A]
```

Figure 2.14 *Example of Descriptors and Description*

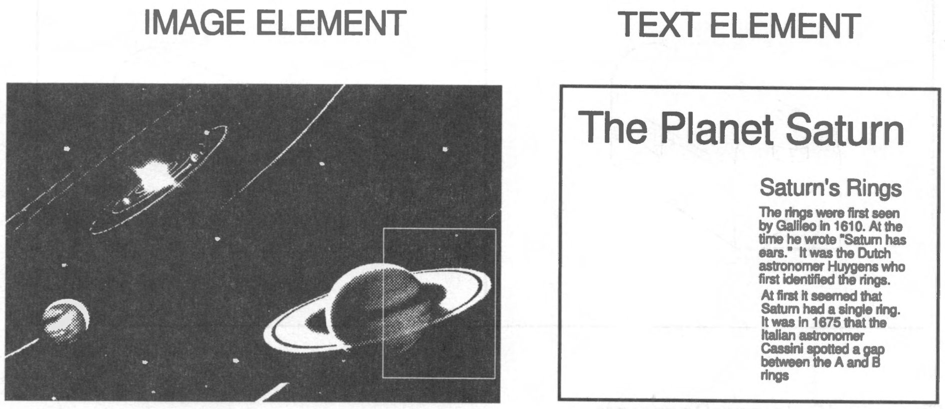
Figure 2.14 is a graphical representation of the description for the object (A) and the translation that results from the SHIFT (x,y) [Z] operator.

## 2.6 The Display File

We have mentioned methods for representing different geometrical image elements so that they can be reproduced and how these techniques can bring substantial savings in the image storage space. We have also shown how stored image data can be manipulated logically and mathematically to generate graphics effects, as well as the use of descriptors and descriptions. The structure that serves to encode graphical images is called the display file.

Since the concepts of descriptors and descriptions are the rational foundation for any modeling scheme, display file design is based on the principles of graphics modeling, mentioned in Section 2.5.5. The first step in display file design is usually determining the general structure of the filing system. The level of complexity of the display file structure should be consistent with the requirements of the system or application. The implementation of a full-featured graphical language requires several logical levels and sublevels of image encoding. A specific application, on the other hand, can do without some of these complications.

The most common elements of the display file are the image file, the image segment, and the descriptors.



The Planet Saturn

Saturn's Rings

The rings were first seen by Galileo in 1610. At the time he wrote "Saturn has ears." It was the Dutch astronomer Huygens who first identified the rings.

At first it seemed that Saturn had a single ring. It was in 1675 that the Italian astronomer Cassini spotted a gap between the A and B rings

Figure 2.15 *Image and Text Elements of an Image File*

### 2.6.1 Image File

The image files are subdivisions of a display file. Each image file encodes a single screen image. For example, the viewport of Figure 2.13 consists of a window that shows the planet Saturn, embedded in another window that contains explanatory text. In this case the image file would contain the instructions and references required for reproducing the entire viewport.

Notice that the image file consists of a set of references and instructions for reproducing a screen image, but that the graphic image data is often stored separately. In the preceding example, the image file references two elements: an image element, which holds the bitmap of a portion of the planet Saturn, and a text element, which holds the alphanumeric strings of data to be displayed. These elements are shown in Figure 2.15.

Storing image and text data separately simplifies making the data available to other images. In Figure 2.13 the partial view of the planet Saturn is a portion of a much larger image stored in the image buffer. In this case the display file need contain only a reference that allows identifying the rectangular tile of the image buffer that is to be used in this particular screen. In addition, the image file contains information describing the transformations, if any, to be performed on the data.

Text elements can be stored in the image file or elsewhere, according to their purpose, complexity, and extension. For example, if the use of text is limited to short messages that are part of the graphic images, the most reasonable approach is to store the text strings in the image file. On the other hand, if the program uses and reuses extensive text areas, it is more efficient to create a central text buffer that can be accessed by any image file.

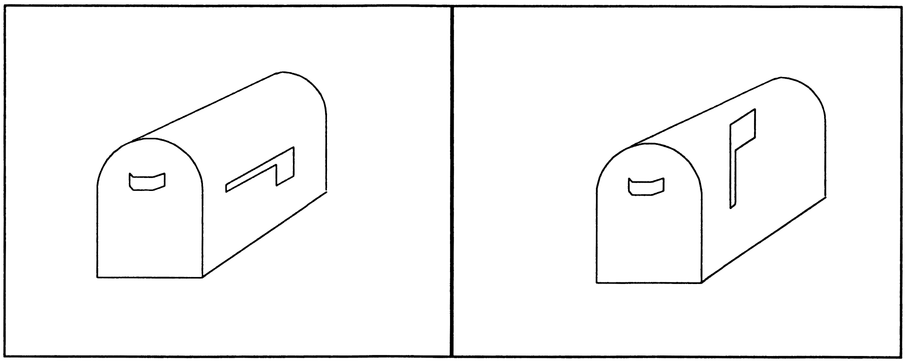


Figure 2.16 *Image Segments*

### 2.6.2 Image Segments

The concept of an image segment is derived from the graphics modeling elements mentioned in Section 2.5.5. An intuitive definition is based on the notion that the segment is a portion of the image that can be considered as a graphic unit. Therefore, the image file can contain more than one image segment. The portion of the image contained in each segment is displayed as a single element.

The image file of Figure 2.16 is composed of two segments: the mailbox and the flag. The mailbox segment is shown in both displays. The flag segment is rotated in the second display.

Most graphic manipulations take place at the level of the image segment.

### 2.6.3 Image Descriptors

The image descriptors are the basic elements of the encoding. In the literature they are also called display file commands, and less appropriately, graphics primitives. The terms descriptors, commands, and primitives all express fundamental properties of this concept. We prefer the term descriptor because the file commands and graphics primitives are used in other contexts in this book.

A descriptor contains all the instructions and data references for displaying a graphical element. The descriptors in Figure 2.14 (move, line, and circle) are used to form the segment (or description) labeled (A). A segment can contain one or more descriptors. For example, the segment for the mailbox in Figure 2.16 requires descriptors for the straight line segments that form the top and bottom of the box and for the arcs that form its ends. The segment for the mailbox flag can contain a single descriptor for a polygon. In designing a graphic system it is generally convenient to first define the image elements and subelements and then provide a descriptor encoding for each element.

The components of a descriptor are the operation code and the operands. The operation code, sometimes called *opcode*, is a mnemonic description of the operations to be performed. The terms move, line, and circle in Figure 2.14 are opcodes. The operands are the data items required by the opcode. In Figure 2.14 the operands follow the opcodes and are enclosed in parentheses or brackets.

# Chapter 3

# Operations on Geometrical Images

## 3.0 Operations on Segments

Segments serve to group image elements so that they can be treated as a single graphic entity. For this reason, the segment is often considered the fundamental unit of graphic operations. In Figure 2.14 we saw how one display is transformed into a second one by changing several image segments. However, these are not the only possible operations that can be performed on segments.

Segment operations are specific to the graphic system and even to the individual implementation. For example, the GKS and the PHIGS standards adopt different views of segments and their structure and implement different segment operations. The segment operations described in the following sections have been chosen because they are the most common ones. The following examples do not conform with any specific graphic standard or language.

### 3.0.1 Creating the Segment

Each segment must be identifiable by the software. This is achieved by assigning a name or identifier to each segment. In the example of Figure 2.14 the operator Dname serves to assign the name A to the segment that depicts a circle enclosed in a square. The translation operation recalls the segment by its name. A graphics system can name segments using any type of symbol or combination of symbols. For instance, in referring to Figure 2.16 we used literal designations for segments, specifically: the mailbox segment and the flag segment. Which method is used to name segments is inconsequential as long as each segment is uniquely identified.

A graphics system must often provide ways for creating new segments. This operation typically assigns a name to the newly created segment. In addition, at creation time the software performs several required checks, specifically:

1. Verify that the segment name is a valid designation according to the system's conventions.
2. Verify that there is sufficient memory space available to create the new segment.
3. Verify that the new segment does not exceed the total number allowed by the system.

The answers to these and possibly other questions determine if the CREATE SEGMENT operation executes successfully or fails. A return code is used to inform the caller of the results. In addition to performing validity checks and to assigning the segment a name, the segment creation operation builds the segment's data structures and initializes the segment variables. Finally, the name of the new segment is added to a list that contains the names (and perhaps some control data) of all the valid segments in the system.

### 3.0.2 Opening and Closing the Segment

A graphical system may be designed so that the CREATE SEGMENT operation automatically opens a segment for input and output. Alternatively, a separate operation may be required for opening the segment. If it must be independently enabled for input and output, the system usually contains OPEN SEGMENT and CLOSE SEGMENT functions. One of the control fields frequently found in the segment's data structure reflects whether the segment is in open or closed status.

Many graphical drawing routines store the input data on the segment currently open. This mode of operation requires that the graphics system implement a mechanism to ensure that only one segment be open at a time.

### 3.0.3 Renaming and Deleting the Segment

The RENAME SEGMENT operation is frequently provided for assigning the segment a new name without altering its contents or status. The DELETE SEGMENT operation erases the segment name from the segment list and frees the memory space occupied by the segment. Both of these operations usually perform a series of consistency checks analogous to those previously listed for the CREATE SEGMENT operation.

## 3.1 Segment Attributes

The segment's characteristics are called *attributes*. The segment's attribute affect the entire segment. Many graphics manipulations performed on segments consist of changing the segment's attributes.

Segment attributes are different in various graphics systems and so are the graphic functions associated with them. The attributes listed in the following sections are those most frequently used. Nevertheless, some graphics systems and applications do not require all of these attributes, while others implement attributes not present in this list.

### 3.1.1 Visibility Attribute

The most elementary segment attribute is its *visibility*. Visibility allows for the modification of an image by controlling whether one or more elements are displayed or not. In Figure 3.1 the segment containing the message PICKUP REQUEST is not visible in image number 1, while the segment containing the message EMPTY is visible. These visibility attributes are inverted in display number 2.

### 3.1.2 Line Color, Fill Color, and Line Style

Color graphics systems implement line color and fill color attributes. The fill color is valid only in reference to closed areas. For instance, a polygon, or even the entire screen background, can be assigned a specific fill color.

The line style attribute can be implemented in color or in monochrome systems. *Computer-assisted design* (CAD) systems, used in engineering and architecture, and drawing programs usually furnish an extensive selection of line style attributes. Some common line styles are solid, dashed, dotted, and dot-dashed.

Line thickness can also be implemented as a line style attribute. Since the thinnest line possible consists of a sequence of adjacent pixels, a thicker line can be specified in terms of its pixel thickness or of any other convenient scale.

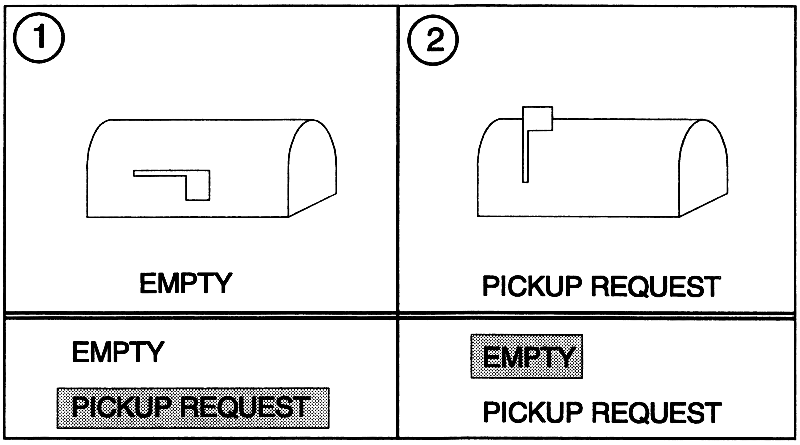


Figure 3.1 *Segment Visibility Attribute*

### 3.1.3 Foreground Priority

Interference conflicts must be resolved when two or more segments are superimposed on the same area of the viewport. These conflicts take place in regard to the predominance of a pattern or color, in the case of superimposed surfaces, or the predominance of an outline, in the case of superimposed wireframe figures. Assigning foreground priorities to each of the segments solves the conflict. Segments with a higher foreground priority than the current segment opaque it, while those with a lower foreground priority are opaqued. Figure 3.2 shows the creation of different images by changing the foreground priorities of its various segments.

The segment foreground priority attribute is usually a numerical value that designates the spatial precedence of each segment. One possible scheme is to assign a priority value of zero to the foreground segment and successive integer priorities to the segments in posterior planes. By examining this value the program can determine the noninterference privilege of each segment, as in Figure 3.2.

## 3.2 Graphical Data Structures

The layout of a storage format for memory-resident graphical data is one of the most laborious phases of the design of a graphic system or application. The following details require careful consideration at design time:

1. The storage formats must be compatible with the programming language or languages used to manipulate the data.

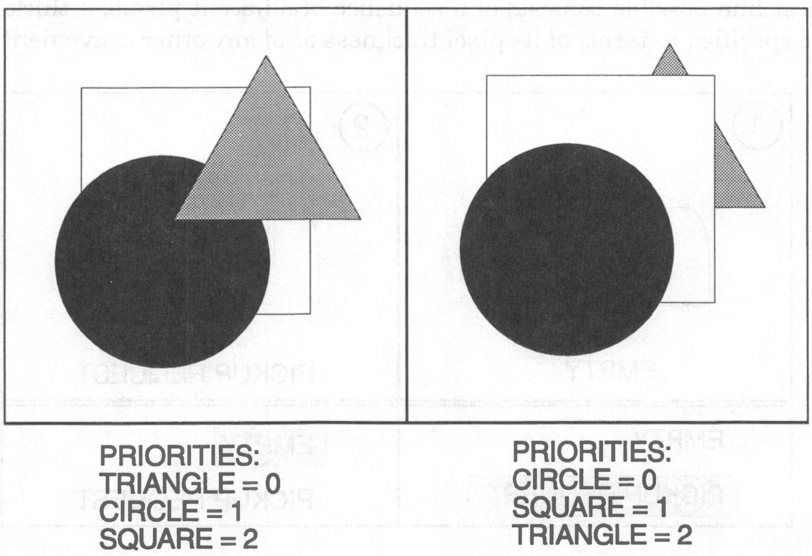


Figure 3.2 *Manipulation of Foreground Priorities*

2. Each data item should be encoded in the most compact format that allows representing the range of values of the variable.
3. Data structures should not be of a predetermined size. The size of the structure should be dynamically determined according to the number of parameters to be stored, giving greater flexibility to the storage system.
4. If feasible, graphics transformations should be performed by means of matrix operations. Such transformations are easier if the data is stored in a matrixlike structure.
5. The designer should consider implementing independent procedures to interface with the data structures. This approach ensures that the processing routines are isolated from the complexities of the storage system. An additional advantage is that only the access routines have to be changed if the data structures are modified during program development, as is so often the case.

### 3.2.1 Display File Elements

The display file consists of one or more images files, the image files contain one or more image segments, and the image segments are formed by descriptors. The structure is shown in Figure 3.3.

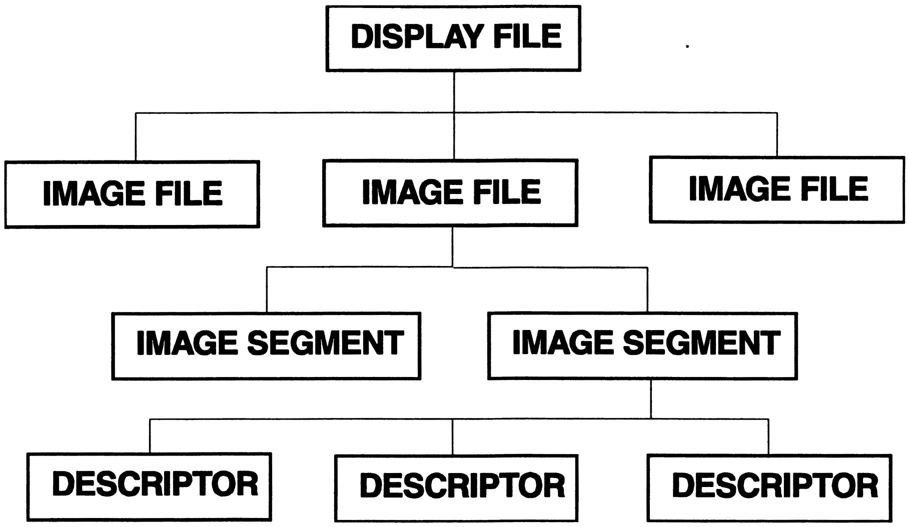


Figure 3.3 *Elements of the Display File*

Considering the hierarchy in Figure 3.3 we can state that the display file is a collection of image files, one for each image element. In this sense the display file is nothing more than a reference table to these image elements. The graphical images are made up of segments. Each image segment is assigned an area of the image file. This segment area, in turn, contains the descriptors of the primitive operations that must be executed in displaying the segment. In the following sections we discuss these individual elements.

## Descriptors

The graphical data is actually stored at the lowest levels of the display file structure, which is that of the descriptors. The descriptors of a particular system correspond to the graphics operations actually implemented. Each descriptor contains an operation identification code (opcode) as well as the necessary numerical operands.

Since the descriptors correspond to graphics primitives of various types, the number of operands varies for different descriptors. For example, the descriptor for a point requires only one set of coordinates (two operands), while the descriptor for a circle requires the coordinates of the origin and the length of the radius (three operands).

In order to display the image segment, the software must be able to find the beginning of each descriptor. Since the operand field may vary in length, the encoding scheme should provide a way to identify the first item of each descriptor. One method is to have a reserved data item for storing the operand count. This data item typically follows the descriptor opcode. Figure 3.4 shows one possible scheme for descriptor encoding.

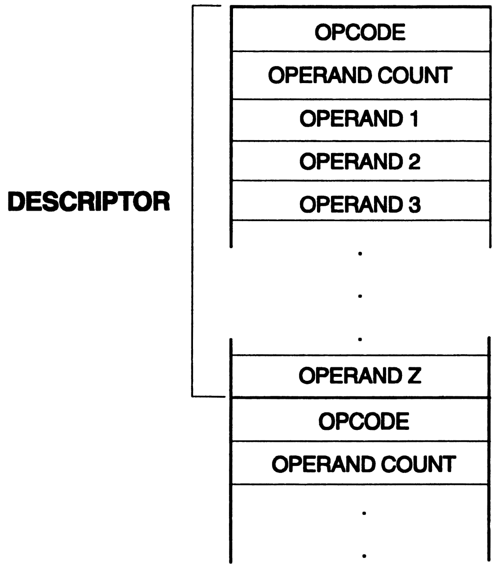


Figure 3.4 *Encoding Scheme for Descriptor*

In the encoding method of Figure 3.4, once the position of the first descriptor is determined, the program can index to the next descriptor using this operand count and the previous descriptor's start address. Notice that the scheme in Figure 3.4 assumes that all entries are of the same length. Another variation that allows for entries of different lengths can be based on storing not the operand count, but its byte length.

The encoding must also provide a means for determining the last descriptor in a segment. One way to do this is by reserving a specific code to signal the end of the segment. For example, opcode FFH could be used for this purpose. Figure 3.5 is a flowchart of the logic required in a routine for executing each descriptor in a segment file.

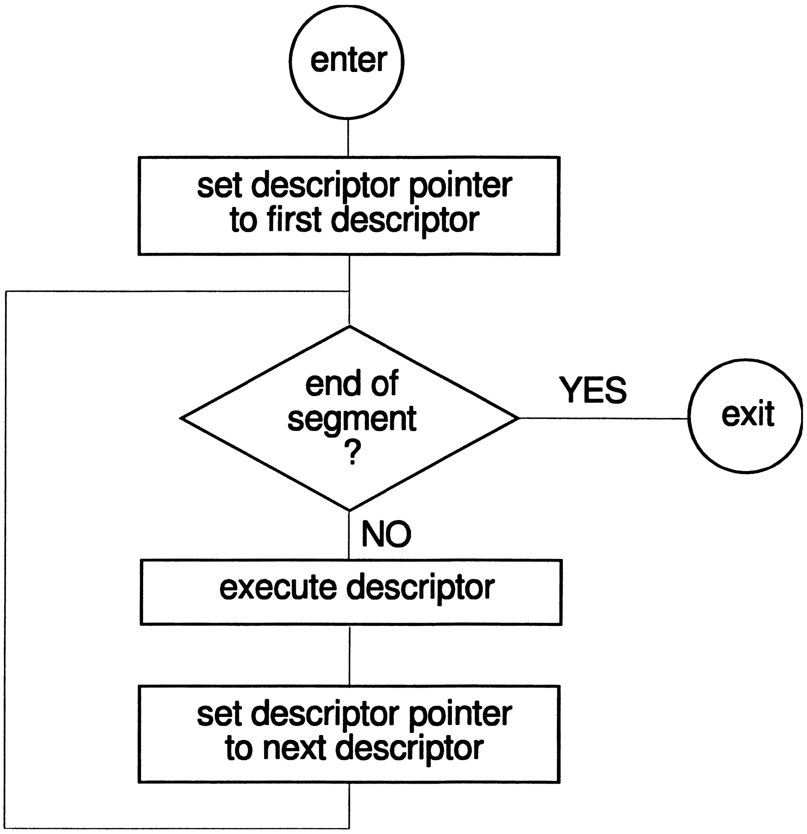

enter
set descriptor pointer to first descriptor
end of segment ?
YES
exit
NO
execute descriptor
set descriptor pointer to next descriptor

Figure 3.5 *Partial Flowchart for Descriptor Processing*

## Image Segments

The image segments contain or reference the segment attributes. Segment attributes affect some or all of the descriptors in a segment; for instance, if the visibility attribute is zero (segment invisible), the descriptors in the segment file are not executed. The processing checks the segment attribute fields before executing the descriptors.

The attributes can be encoded in the same data area as the descriptors or in a separate area that holds the attributes of each segment in the display file. In the first option, the encoding scheme for the segment attributes follows a pattern similar to that of the descriptors. Each attribute field contains an attribute code, followed by an operand count field, and the attribute operands. It is usually preferable to reserve a certain numerical range for the descriptor opcodes and another one for the attribute codes. For example, values between 1 and 99 can be used for the descriptors and values between 99 and 199 for the attributes. Thus, the output routine can easily identify either encoding, even if descriptors and attributes are mixed in the segment file.

Table 3.1 is a possible descriptor encoding scheme.

Table 3.1 *Sample Descriptor Encoding*

```
              Data items                      Displacement
DESCRIPTOR ----------->|=================|
                       |     opcode      |      = 0
                       |-----------------|
                       |  operand count  |      = 1
                       |-----------------|
                       |    operand 1    |      = 2
                       |-----------------|
                       |    operand 2    |      = 3
                       |-----------------|
                       |    operand 3    |      = 4
                       |-----------------|
                                .
                                .
                                .
                       |-----------------|
                       |  last operand   |      = operand count
DESCRIPTOR ----------->|=================|
                       |     opcode      |      = 0
                       |-----------------|
                       |  operand count  |      = 1
                       |-----------------|
                                .
                                .
```

When the attributes are held in a separate data area, it is usually called a *segment table*. The segment table contains the segment identification code as well as the values for the different attributes. In some encoding schemes the segment table can also hold certain segment parameters. One advantage of this method is that keeping the attributes separate from the segment data makes it possible to reuse the same segment file in several images, and at the same time, preserve the encoding for each image.

Figure 3.6 shows the geometrical elements that form the mailbox image segment. The illustration assumes that descriptor opcode 2 represents a straight line and opcode 7 an elliptical arc.

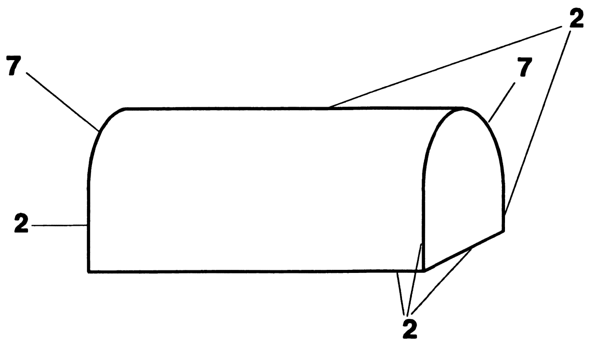


Figure 3.6 *Straight Lines (2) and Ellipses (7) in the Mailbox Segment*

Notice that there are a total of six straight lines and two ellipses in the mailbox segment in Figure 3.6. The image file in Table 3.2 shows the segment attributes associated with Figure 3.6.

Table 3.2 *Schematic Segment Encoding for Mailbox in Figure 3.6*

```
                        SEGMENT A [mail box]
                   |    |2|4|o|o|o|o|
                   |    |2|4|o|o|o|o
                   |    |2|4|o|o|o|o
                   |    |2|4|o|o|o|o
Descriptors -----  |    |2|4|o|o|o|o
                   |    |2|4|o|o|o|o
                   |    |7|6|o|o|o|o|o|o|
                   |    |7|6|o|o|o|o|o|o|
                   |    |0|0|
                        | |  |---------|
                        | |       |_________ Operands
                        | |
                        | |_________________ Operand count
                        |
                        |___________________ Opcodes
                                             2 = straight line
                                             7 = ellipse
                                             0 = end of segment
Legend:
o = operands
```

In Table 3.2 the operands are represented by the letter "o." In a real application these operands are the coordinates and other parameters necessary for describing the individual geometrical elements.

Notice in Figure 3.1 that the same segment file can be used for both versions of the mailbox flag. In this case, the segment table entry for image number 1 does not specify a rotation transformation, while the segment table entry for the mailbox flag in image number 2 specifies a 90-degree counterclockwise rotation using the lower-left vertex of the polygon as a center of rotation, or pivot point. The visibility attribute for the text segments in these images is handled in a similar manner.

## Image Files

An image can be described graphically by the segments that form it. Therefore, the *image file* is a list of segment files. The linking of these segments can be accomplished in several ways. One possible encoding scheme is to consider the image file as a supersegment, with an identifying name and all the component segments placed consecutively within the image data area. A data position in the image file is reserved for the segment count. This allows the processing routine to form the image by executing the individual segment operations until the count is exhausted. The segment count data item also allows the software to index from image file to image file. Table 3.3 shows a possible implementation of the encoding for an image file corresponding with image number 1 of Figure 3.1. Notice that the scheme adopted in this sample uses a segment table as part of the image file.

Table 3.3 *Sample of Image File Encoding*

```
                    Segment count ____________
                                              |
IMAGE FILE 1 [Image No. 1 in Figure 3.1]|4|
  SEGMENT A [mail box]
  SEGMENT B [mail box flag]
  SEGMENT C [message "PICKUP REQUEST"]
  SEGMENT D [message "EMPTY"]
  |-------------------------------------|
  |      SEGMENT ATTRIBUTES TABLE       |
  |-------------------------------------|
   SEGMENT A --- |110|1|o|
                  |150|1|1|
   SEGMENT B --- |110|1|o|
                  |120|4|o|o|o|o|
                  |150|1|0|
   SEGMENT C --- |100|1|0|
   SEGMENT D --- |100|1|1|
                  |256|0|
                    |   ||-----|
                    |   |     |
                    |   |     |_______ Operands
                    |   |
                    |   |____________ Operand count
                    |
                    |___ Attribute code
                         110 = line style
                         150 = foreground priority
                         120 = rotation
                         100 = visibility
                         256 = end of table
```

## 3.3 Image Transformations

Certain image changes can be made by performing mathematical operations on its coordinate points. Figure 3.7 shows the translation of a line from coordinates (2,2) and (10,14) to coordinates (10,2) and (18,14).

Notice that in Figure 3.7 the translation is performed by adding 8 to the start and end $x$ coordinates of the original line. This operation on the $x$ axis performs a horizontal translation. A vertical translation is performed by operating on the $y$ coordinate. By the same token, to translate the line both horizontally and vertically, the program operates arithmetically on both coordinate axes.

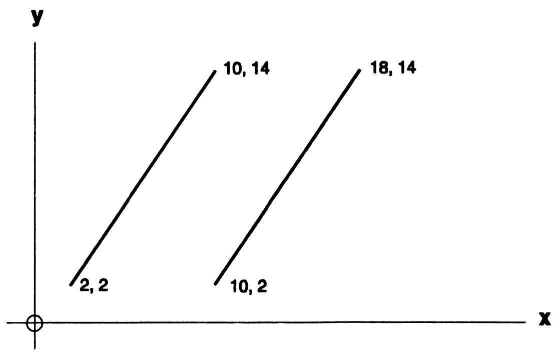


Figure 3.7 *Translation of a Straight Line*

In practice, the mathematical manipulations are performed on the data structures contained in the image file. Therefore, the design of these data structures determines the degree of ease or difficulty with which these operations are performed by the software.

In addition to organizing image data in structures that facilitate the mathematical transformations, graphics software must also provide the processing logic to perform the necessary operations. Both elements, image data structures and computational logic, determine the image transformation facilities of a graphics system or application.

### 3.3.1 The Coordinates Matrix

A matrix is a set of values arranged in a rectangular array. Each value in the array is called an element of the matrix. In the context of graphical programming, matrices are often used to hold coordinate points. This form of storing graphical data allows using linear algebra to perform transformations. Figure 3.8 shows the approximate location of seven stars of the constellation Ursa Minor, also known as the Little Dipper. The individual stars are labeled with the letters a through g. The star labeled "a" corresponds to Polaris (the North Star).

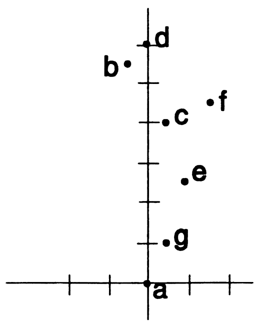


Figure 3.8 *Stars of the Constellation Ursa Minor (Little Dipper)*

The following matrix holds the coordinates of the stars in Figure 3.8.

```
                    Coordinates
                    x       y
Star a ........     0       0
     b ........    -1      11
     c ........     1       8
     d ........     0      12
     e ........     2       5
     f ........     3       9
     g ........     1       2
```

In two-dimensional systems, the coordinates matrix is formed by sets of $x$ and $y$ coordinates, as in the above case. In three-dimensional systems, the coordinate matrix holds the $x$, $y$, and $z$ coordinates. The following matrix represents the coordinate points for a line in three-dimensional space.

```
                          Coordinates
                  x           y           z
start point -->   2           7          12
  end point -->   4          10          24
```

The following sections explain the fundamental matrix operations that are most useful in graphics and animation programming. The reader familiar with matrices and elementary matrix arithmetic can skip to Section 3.5.

## 3.4 Matrix Arithmetic

Matrices are used in many fields of mathematics. In linear algebra they are used to hold the coefficients of linear equations. The equations can be manipulated (and often solved) by performing operations on the rows and columns of the matrix. One approach to solving a system of linear equations, known as Gauss-Jordan elimination, consists of several processing steps that convert the matrix to a special configuration called the *reduced row-echelon form*. Once in this form, the system can be solved by inspection.

Matrix operations are convenient in performing the primitive transformations of translation, rotation, and scaling that are common in graphics and animation programming. In order to derive the rules of matrix arithmetic, we must first define the matrix and its component elements. We have already seen that a matrix is a rectangular array of numbers. As is customary, in the following sections we use capital letters to represent matrices. For example, the following matrix, designated by the letter A, has three rows and two columns.

$$A = \begin{vmatrix} 10 & 22 \\ 3 & 4 \\ 7 & 1 \end{vmatrix} \tag{1}$$

The size of a matrix is the number of rows and columns that it contains. The usual practice is to state matrix size as a product of rows by columns. For example, matrix A, in Example (1), is a 3-by-2 matrix.

### 3.4.1 Scalar-by-Matrix Operations

An individual numerical quantity is called a scalar. Scalar-by-matrix operations are the simplest procedures of matrix arithmetic. Example (2) shows the multiplication of matrix A by the scalar 3.

$$3A = \begin{vmatrix} 30 & 66 \\ 9 & 12 \\ 21 & 3 \end{vmatrix} \tag{2}$$

If a scalar is represented by the variable *s*, the product matrix *s*A is the result of multiplying each element in the matrix A by the scalar *s*. By the same token, scalar addition and subtraction are obtained by adding or subtracting the scalar quantity to or from each matrix element.

### 3.4.2 Matrix Addition and Subtraction

Matrix addition and subtraction are performed by adding or subtracting each element in a matrix to or from the corresponding element of another matrix of equal size. Example (3) shows matrix addition. Matrix C is the algebraic sum of each element in matrices A and B.

$$A \begin{vmatrix} 2 & 4 \\ 3 & 11 \\ 1 & 5 \\ 1 & -1 \end{vmatrix} + B \begin{vmatrix} 1 & 2 \\ 2 & 2 \\ -1 & -3 \\ 0 & 0 \end{vmatrix} = C \begin{vmatrix} 3 & 6 \\ 5 & 13 \\ 0 & 2 \\ 1 & -1 \end{vmatrix} \tag{3}$$

The fundamental restriction of matrix addition and subtraction is that both matrices must be of equal size; that is, they must have the same number of rows and of columns. Matrices of different sizes cannot be added.

### 3.4.3 Matrix Multiplication

The operation of matrix addition intuitively corresponds to conventional addition; that is, the elements of two matrices are added to obtain the sum. Matrix multiplication, on the other hand, is not the multiplication of the corresponding elements of two matrices, but a unique sum-of-products operation.

In matrix multiplication the elements of a row in the multiplicand matrix are multiplied by the elements in a column of the multiplier matrix. These products are then added to form the products matrix. The process is easily understood through an illustration of the steps involved. Consider the matrices in Example (4).

From the definition of matrix multiplication it can be deduced that if the rows of the first matrix are multiplied by the columns of the second matrix each row of the multiplier must have the same number of elements as each column of the multiplicand. Notice that in Example (4) the product A × B meets this requirement. Also note that the product B × A is not possible, since matrix B has three elements per row and matrix A has only two elements in each column. Therefore, in Example (4), the matrix operation A × B is possible but B × A is undefined. The row by column operation in A × B is performed as follows:

$$A = \begin{vmatrix} 1 & 3 & 5 \\ 2 & 1 & 0 \end{vmatrix} \qquad B = \begin{vmatrix} 5 & 10 & 2 \\ 1 & 2 & 3 \\ 11 & 5 & 4 \end{vmatrix} \tag{4}$$

The products matrix has the same number of rows as the multiplicand matrix and the same number of columns as the multiplier matrix. In example (4) the products matrix C has the same number of rows as A and the same number of columns as B. In other words, C is a 2 × 3 matrix. The elements obtained by the above operations appear in matrix C in the following manner:

```
   First
 Row of A        Columns of B        Products        Sum
 1   3   5   *    5   1   11   =   5  +  3  +  55  =  63
 1   3   5   *   10   2    5   =  10  +  6  +  25  =  41
 1   3   5   *    2   3    4   =   2  +  9  +  20  =  31

  Second
 Row of A        Columns of B        Products        Sum
 2   1   0   *    5   1   11   =  10  +  1  +   0  =  11
 2   1   0   *   10   2    5   =  20  +  2  +   0  =  22
 2   1   0   *    2   3    4   =   4  +  3  +   0  =   7
```

In the course of developing Example (4) we commented that the operation A × B is possible but that B × A is undefined since matrix multiplication is not commutative. Therefore, the product of two matrices could be different if the matrices were taken in different order. In fact, regarding nonsquare matrices, it can be stated that if A × B is defined, B × A is undefined.

$$C = \begin{vmatrix} 63 & 41 & 31 \\ 11 & 22 & 7 \end{vmatrix}$$

Matrix multiplication is associative. Therefore, the product of three or more matrices is equal no matter the order in which they are multiplied. For example, (A × B) × C equals A × (B × C). In performing graphics transformations we find use for the associative and the noncommutative properties of matrix multiplication.

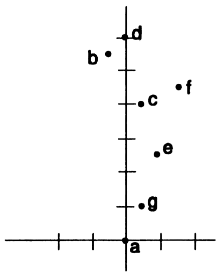
## 3.5 Geometrical Transformations

A geometrical transformation is the conversion of one image into another one by performing a mathematical operation on its coordinate points. Geometrical transformations are simplified if the image's coordinates are stored in a rectangular array called a matrix. In the following sections, we describe the most common transformations: translation, scaling, and rotation. The transformations are first described in terms of matrix addition and multiplication, and later standardized so that they can all be expressed in terms of matrix multiplications.

### 3.5.1 Translation

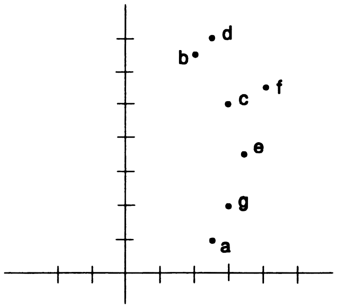
Translation is the movement of a graphical object to a new location by adding a constant value to each coordinate point that defines the object. The operation requires that a constant be added to all the coordinates in each plane, but the constants can be different for each plane. For example, a translation takes place if the constant 5 is added to all $x$ coordinates and the constant 2 to all $y$ coordinates of an object represented in a two-dimensional plane.

In the top part of Figure 3.9 we see the graph and matrix of seven stars in the constellation Ursa Minor. A translation transformation is performed by adding 5 to the $x$ coordinate of each star and 2 to the $y$ coordinate. The bottom part of Figure 3.9 shows the translated image and the new coordinates.



original coordinates:

| star | x | y |
|---|---|---|
| a | 0 | 0 |
| b | -1 | 11 |
| c | 1 | 8 |
| d | 0 | 12 |
| e | 2 | 5 |
| f | 3 | 9 |
| g | 1 | 2 |



translated coordinates (x+5, y+2):

| star | x | y |
|---|---|---|
| a | 5 | 2 |
| b | 4 | 13 |
| c | 6 | 10 |
| d | 5 | 14 |
| e | 7 | 7 |
| f | 8 | 11 |
| g | 6 | 4 |

Figure 3.9 *Translation Transformation*

In terms of matrix operations, the translation can be viewed as follows:

| Original coordinates matrix A | | Transformation matrix B | | Transformed coordinates matrix C | |
|---|---|---|---|---|---|
| *x* | *y* | *x* | *y* | *x* | *y* |
| 0 | 0 | 5 | 2 | 5 | 2 |
| -1 | 11 | 5 | 2 | 4 | 13 |
| 1 | 8 | 5 | 2 | 6 | 10 |
| 0 | 12 | 5 | 2 | 5 | 14 |
| 2 | 5 | 5 | 2 | 7 | 7 |
| 3 | 9 | 5 | 2 | 8 | 11 |
| 1 | 2 | 5 | 2 | 6 | 4 |

This can also be expressed as

$$A + B = C$$

where A represents the original coordinates matrix, B the transformation matrix, and C the matrix holding the transformed coordinates.

Notice that the transformation matrix holds the constants to be added to the *x* and *y* coordinates. Since, by definition of the translation transformation, the same value must be added to all the elements of a coordinate plane, it is evident that the columns of the transformation matrix always hold the same numerical value.

## 3.5.2 Scaling

To scale is to apply a multiplying factor to the linear dimension of an object. A *scaling* transformation is the conversion of a graphical object into another one by multiplying each coordinate point that defines the object. The operation requires that all the coordinates in each plane be multiplied by the scaling factor, although the scaling factors can be different for each plane. For example, a scaling transformation takes place when all the *x* coordinates of an object represented in a two-dimensional plane are multiplied by 2 and all the *y* coordinates of this same object are multiplied by 3. In this case the scaling operation is said to be asymmetrical.

By comparing the definition of the scaling transformation to that of the translation transformation we notice that translation is performed by adding a constant value to the coordinates in each plane, while scaling requires multiplying these coordinates by a factor. In fact, the scaling transformation can be represented in matrix form by taking advantage of the properties of matrix multiplication.

Figure 3.10 shows a scaling operation of a square into a rectangle.

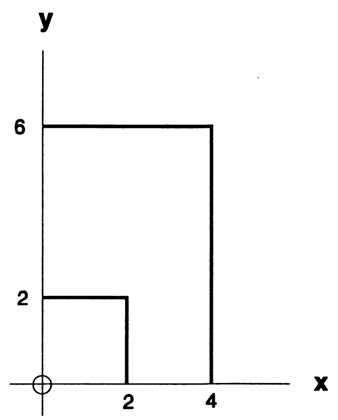


Figure 3.10 *Scaling Transformation*

The coordinates of the square in Figure 3.10 can be stored in a 4-by-2 matrix, as follows:

$$\begin{array}{r|cc|}
 & \multicolumn{2}{c}{\text{Coordinates}} \\
 & x & y \\
\text{start point} & 0 & 0 \\
 & 2 & 0 \\
 & 2 & 2 \\
\text{end point} & 0 & 2
\end{array}$$

The transformation matrix holds the factors that must be multiplied by the *x* and *y* coordinates in order to perform the transformation. Using the letters S*x* to represent the scaling factor for the *x* coordinates, and the letters S*y* to represent the scaling factor for the *y* coordinates, the scaling transformation matrix can be expressed as follows:

$$\begin{vmatrix} Sx & 0 \\ 0 & Sy \end{vmatrix}$$

The transformation of Figure 3.10, which converts the square into a rectangle, can be represented in matrix form as follows:

$$\begin{array}{c}\text{Original}\\\text{coordinates}\\\text{matrix}\end{array}
\begin{vmatrix} x & y \\ 0 & 0 \\ 2 & 0 \\ 2 & 2 \\ 0 & 2 \end{vmatrix}
*
\begin{array}{c}\text{Scaling}\\\text{matrix}\\ Sx \quad Sy \\ \begin{vmatrix} 2 & 0 \\ 0 & 3 \end{vmatrix}\end{array}
=
\begin{array}{c}\text{Transformed}\\\text{coordinates}\\\text{matrix}\end{array}
\begin{vmatrix} x & y \\ 0 & 0 \\ 4 & 0 \\ 4 & 6 \\ 0 & 6 \end{vmatrix}$$

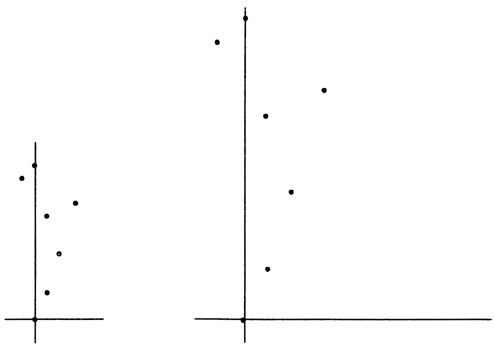


Figure 3.11 *Symmetrical Scaling (Zooming)*

The intermediate steps in the matrix multiplication operation can be obtained following the rules of matrix multiplication described in Section 3.4.3.

Figure 3.11 shows the scaling transformation of the graph of the constellation Ursa Minor. In this case, in order to produce a symmetrical scaling, the multiplying factor is the same for both axes. A symmetrical scaling operation is sometimes referred to as a *zoom*.

## 3.5.3 Rotation

A *rotation* is the conversion of a graphical object into another one by moving all coordinate points that define the original object, by the same angular value, along circular arcs with a common center. The angular value is called the *angle of rotation,* and the fixed point that is common to all the arcs is called the *center of rotation*. Observe that some geometrical figures are unchanged by specific rotations. For example, a circle is unchanged by a rotation about its center, and a square is unchanged if it is rotated by an angle that is a multiple of 90 degrees. In the case of a square the intersection point of both diagonals is the center of rotation.

The mathematical interpretation of the rotation is obtained by applying elementary trigonometry. Figure 3.12 shows the counterclockwise rotation of points located on the coordinate axes at unit distances from the center of rotation.

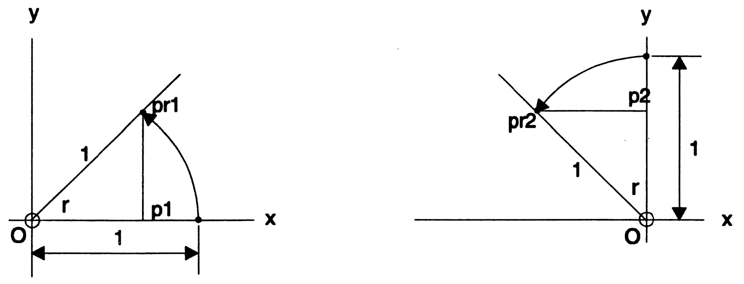


Figure 3.12 *Rotation of a Point*

On the left side of Figure 3.12, point $p1$, with coordinates (1,0), is rotated counterclockwise through an angle $r$. The coordinates of the rotated point ($pr1$) can be determined by solving the triangle with vertices at O, $p1$, and $pr1$, as follows:

```
cos r = x/1, therefore x = cos r
sin r = y/1, therefore y = sin r
```

The coordinates of the rotated point $pr2$, on the right side of Figure 3.12, can be determined by solving the triangle with vertices at O, $p2$, and $pr2$.

```
sin r = -x/1, therefore x = - sin r
cos r = y/1, therefore y = cos r
```

The coordinates of the rotated points can now be expressed as follows:

```
coordinates of pr1 = (cos r, sin r)
coordinates of pr2 = (-sin r, cos r)
```

From these equations we can derive a transformation matrix, which, through matrix multiplication, yields the new coordinates for the counterclockwise rotation through an angle A:

$$\begin{vmatrix} \cos r & \sin r \\ -\sin r & \cos r \end{vmatrix}$$

We are now ready to perform a rotation transformation through matrix multiplication. Figure 3.13 shows a clockwise rotation, through an angle of 60 degrees, with the center of rotation at the origin of the coordinate axes.

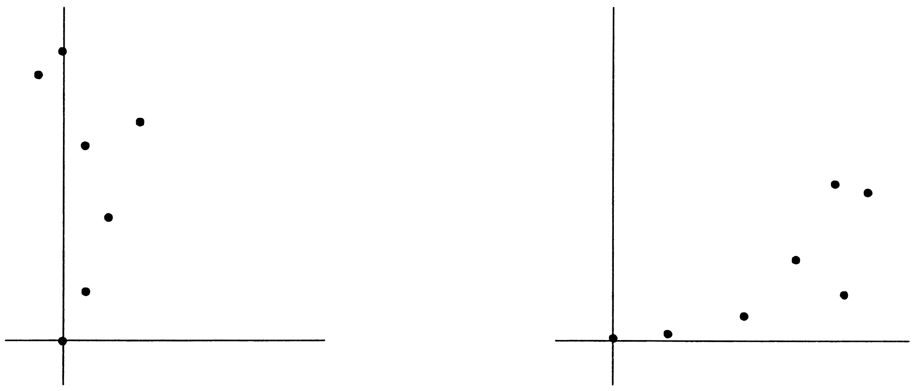


Figure 3.13 *Rotation Transformation*

The coordinates of the original polygon lines can be stored in a 4-by-2 matrix as follows:

$$\begin{array}{lcc} & \multicolumn{2}{c}{\text{Coordinates}} \\ & x & y \\ p1 \dashrightarrow & 10 & 2 \\ p2 \dashrightarrow & 12 & 0 \\ p3 \dashrightarrow & 14 & 2 \\ p4 \dashrightarrow & 12 & 4 \end{array}$$

We have seen that the transformation matrix for clockwise rotation through an angle *r* is

$$\begin{vmatrix} \cos r & \sin r \\ -\sin r & \cos r \end{vmatrix}$$

Evaluating this matrix for a 60-degree rotation results in the following trigonometric functions:

$$\begin{vmatrix} 0.5 & 0.867 \\ -0.867 & 0.5 \end{vmatrix}$$

The rotation can now be expressed as a product of two matrices.

$$\begin{array}{l} \\ \\ \\ \\ p1 \dashrightarrow \\ p2 \dashrightarrow \\ p3 \dashrightarrow \\ p4 \dashrightarrow \end{array}
\begin{array}{c} \text{Original} \\ \text{polygon} \\ \text{coordinates} \\ \begin{array}{cc} x & y \end{array} \\ \begin{vmatrix} 10 & 2 \\ 12 & 0 \\ 14 & 2 \\ 12 & 4 \end{vmatrix} \end{array}
*
\begin{array}{c} \text{Rotation matrix} \\ \text{60 degrees} \\ \text{clockwise} \\ \\ \begin{vmatrix} 0.5 & 0.867 \\ -0.867 & 0.5 \end{vmatrix} \end{array}
=
\begin{array}{c} \text{Rotated} \\ \text{polygon} \\ \text{coordinates} \\ \begin{array}{cc} x & y \end{array} \\ \begin{vmatrix} 3.87 & 9.87 \\ 6 & 10.4 \\ 5.27 & 13.4 \\ 2.53 & 12.4 \end{vmatrix} \end{array}
\begin{array}{l} \\ \\ \\ \\ \leftarrow pr1 \\ \leftarrow pr2 \\ \leftarrow pr3 \\ \leftarrow pr4 \end{array}$$

The intermediate steps in the matrix multiplication operation are obtained following the rules of matrix multiplication described in Section 3.4.3.

### 3.5.4 Homogeneous Coordinates

Translation, scaling, and rotation can be expressed mathematically in terms of matrix operations; this method allows a more efficient approach to graphical transformations. The one inconsistency in the method described is that rotation and scaling are expressed in terms of matrix multiplication while translation is expressed as matrix addition.

By means of a simple artifice it is possible to represent the translation transformation as matrix multiplication. This scheme requires adding a dummy parameter to the coordinates matrices and expanding the transformation matrices to 3-by-3 elements. However, it simplifies processing by allowing all three transformations to be performed by means of a single matrix operation.

The following example shows the necessary manipulations.The coordinates of a point can be expressed in the following matrix.

```
                Coordinates
                 x       y
point -->      | 5       2 |
```

This matrix can be expanded to three rows by using a dummy matrix parameter, labeled $w$. Notice that if $w$ is not to affect coordinates $x$ and $y$ in two-dimensional transformations, it must meet the following requirement.

$$x = x * w, \qquad y = y * w$$

Therefore, the only value that can be assigned to $w$ that meets the above condition is 1, which gives us the following matrix:

```
               Coordinates
             x       y     w
point --> |  5       2     1 |
```

We can use the terms T$x$ and T$y$ to represent the horizontal and vertical units of a translation. Using homogeneous coordinates, a transformation matrix for the translation operation can be expressed as follows:

```
      Translation
    transformation
        matrix
  | 1      0      0 |
  | 0      1      0 |
  | Tx     Ty     1 |
```

We test these results by performing a translation by eight units in the horizontal direction (T$x$ = 8) and zero units in the vertical direction (T$y$ = 0) of the point located at coordinates (5,2). The matrix multiplication is as follows:

```
                 | 1  0  0 |     [ 5 + 0 + 8 ] = 13
[ 5  2  1 ] *    | 0  1  0 |  =  [ 0 + 2 + 0 ] =  2 = [ 13  2  1 ]
                 | 8  0  1 |     [ 0 + 0 + 1 ] =  1
```

This operation shows the point at $x = 5, y = 2$ translated eight units to the right, with destination coordinates of $x = 13, y = 2$. The reader should note that the $w$ parameter, set to 1 in the original matrix, remains the same in the final matrix. In practical processing the parameter can be ignored.

### 3.5.5 Concatenation

In order to take full advantage of the system of homogeneous coordinates we must express all the transformation matrices in terms of 3-by-3 matrices. Using homogeneous coordinates, the translation transformation can be expressed in the following matrix:

$$\begin{array}{c}\text{Translation}\\ \text{transformation}\\ \text{matrix}\\ \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{vmatrix}\end{array}$$

The scaling transformation matrix can also be expanded to a 3-by-3 matrix as follows:

$$\begin{array}{c}\text{Scaling}\\ \text{transformation}\\ \text{matrix}\\ \begin{vmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{vmatrix}\end{array}$$

At the same time, the translation transformation matrix for a counterclockwise rotation through an angle *r* can be converted to homogeneous coordinates as follows:

$$\begin{array}{c}\text{Rotation}\\ \text{transformation}\\ \text{matrix}\\ \begin{vmatrix} \cos r & \sin r & 0 \\ -\sin r & \cos r & 0 \\ 0 & 0 & 1 \end{vmatrix}\end{array}$$

Notice that this rotation transformation assumes that the center of rotation is at the origin of the coordinate system.

Matrix multiplication is associative. This means that the product of three or more matrices is equal, no matter which two matrices are multiplied first. By virtue of this property, we are now able to express a complex transformation by combining several basic transformations. This process is generally known as *matrix concatenation*.

A rotation transformation can use any arbitrary point in the coordinate system as a pivot point. For example, in Figure 3.14 polygon number 1 is rotated counterclockwise 90 degrees using point *pa* as a pivot point. Furthermore, to rotate the polygon about any arbitrary point *pa*, the following sequence of transformations can be executed:

1. Translate the polygon so that point *pa* is at the coordinate origin.
2. Rotate the polygon.
3. Translate the polygon so that point *pa* returns to its original position.

In matrix form the sequence of transformations can be expressed as the following product:

$$\begin{array}{c} 1. \\ \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -Tx & -Ty & 1 \end{vmatrix} \end{array} * \begin{array}{c} 2. \\ \begin{vmatrix} \cos r & \sin r & 0 \\ -\sin r & \cos r & 0 \\ 0 & 0 & 1 \end{vmatrix} \end{array} * \begin{array}{c} 3. \\ \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx & Ty & 1 \end{vmatrix} \end{array}$$

Performing the indicated multiplication yields the matrix for a counterclockwise rotation, through angle *r*, about point *pa*, with coordinates (T*x*,T*y*).

$$\begin{vmatrix} \cos r & \sin r & 0 \\ -\sin r & \cos r & 0 \\ -Tx \cos r + Ty \sin r & -Tx \sin r - Ty \cos r + Ty & 1 \end{vmatrix}$$

While matrix multiplication is associative, it is not commutative. The order in which the operations are performed can affect the results. A fact that confirms the validity of the matrix representation of graphic transformations is that, graphically, the results of performing transformations in different sequences can also yield different results. For example, the image resulting from a certain rotation, followed by a translation transformation, may not be identical to the one resulting from performing the translation first and then the rotation.

Figure 3.14 shows a case in which the order of the transformations determines a difference in the final object.

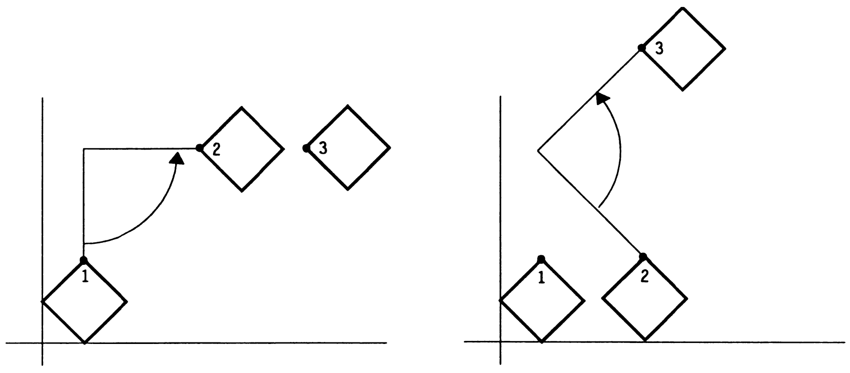


Figure 3.14 *Order of Transformations*

## 3.6 Image Transformations in Animation

Graphic transformations provide a convenient technique for creating consecutive images of a geometrical object. If the consecutive images obey certain physical laws, and if they are projected and erased at sufficient speed, they can be used to create an illusion of movement or change.

Due to image retention the animated images must be flashed at a minimum rate of 24 per second to produce a realistic effect. We have also seen that even with images of moderate complexity, the task of creating and displaying them at this rate can impose an extremely large processing load on the graphics system. Therefore, in animation programming every device or stratagem that improves graphics performance is critically important to the final effect. Performing the image transformation by mathematically operating on matrices of coordinate points saves considerable processing time and effort.

### 3.6.1 Translation, Rotation, and Scaling Animation

The simplest and probably the most used transformation in computer animation is translation. For example, if the pixels along a consecutive path are rapidly illuminated and turned off, the viewer perceives the effect of a dot moving across the screen. By the same token, if all the dots that form a graphical object are consecutively illuminated and turned off along a certain path, the object appears to move across the screen. Figure 3.15 represents a few translations of the image of a spaceship moving across a bit-mapped background.

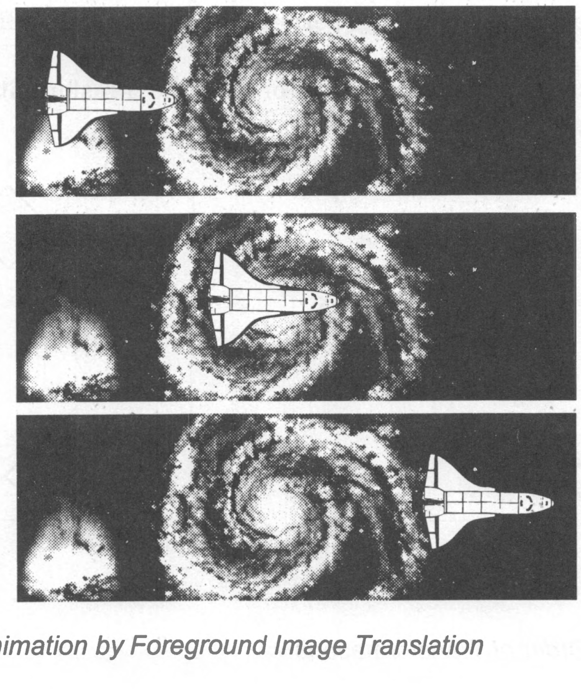


Figure 3.15 *Animation by Foreground Image Translation*

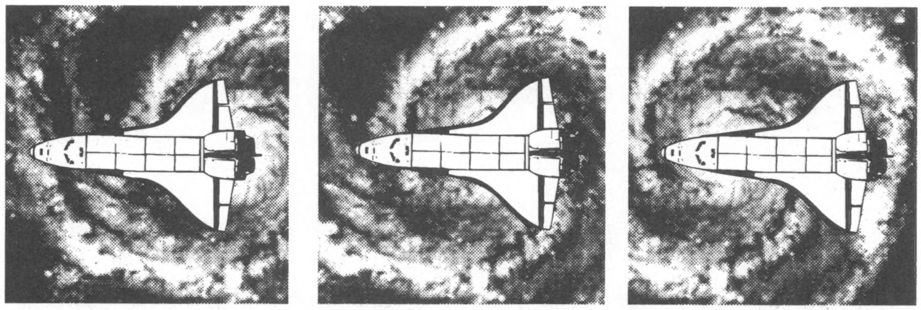


Figure 3.16 *Animation by Background Translation*

In Figure 3.15 the image of the space shuttle, which is a geometrical segment, is translated over the bitmap that represents a spiral nebula. The foreground priority of the space shuttle is higher than that of the background. An alternative manipulation consists of translating the background image while letting the foreground image occupy a fixed position in the viewport. This case is shown in Figure 3.16.

### 3.6.2 Complex Animation

It is also possible to combine more than one transformation in the creation of more refined animation effects. For example, by combining translation and rotation transformations, a wheel appears to roll on the screen. Or, by combining translation and scaling transformations, an object disappears into the background. Figure 3.17 shows the application of translation, scaling, and rotation transformations on the image of the space shuttle to simulate its being drawn into the background nebula. The effect could be enhanced by applying additional transformations to the background image.

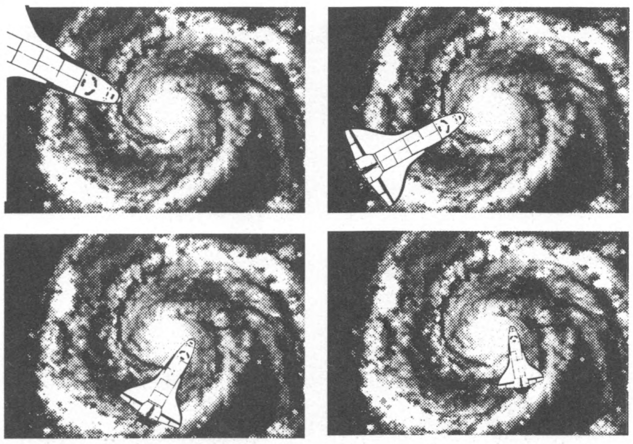


Figure 3.17 *Animation by Translation, Scaling, and Rotation*

# Chapter 4

# Bitmap Image Acquisition and Encoding

## 4.0 Pixel-Coded Image Data

Chapters 2 and 3 were mainly devoted to the encoding, storage, and manipulation of geometrical images, although bitmaps were mentioned incidentally. However, bit-mapped images are as important to the animation programmer as are vector-based ones. This chapter describes the various techniques and standards used in encoding computer graphics images into units of memory storage. It includes a discussion of two popular image data storage formats: Compuserve's GIF and Aldus Corporation's TIFF format.

Bit-mapping is the graphics technique by which one or more memory bits represent the attribute of a screen pixel. The simplest bitmap scheme is to make a memory bit represent a single screen pixel: if the memory bit is set, so is the pixel. However, a graphics image can be encoded in a more complete and efficient structure than is offered by a pixel-by-pixel attribute list.

The movement toward the standardization of image file encodings originated with commercial software developers in need of methods for storing and displaying graphics images. Currently there are over 20 different image file encodings in frequent use. Graphics applications often import or export images encoded in over a dozen file formats. Although some of these commercial encodings have gained more popularity than others, very little has been achieved in standardizing image file encodings. In this chapter we have selected the two image file formats that we believe are more useful and that have gained more widespread acceptance in the field. This selection does not imply that we endorse these particular encodings or approve of their design or operation.

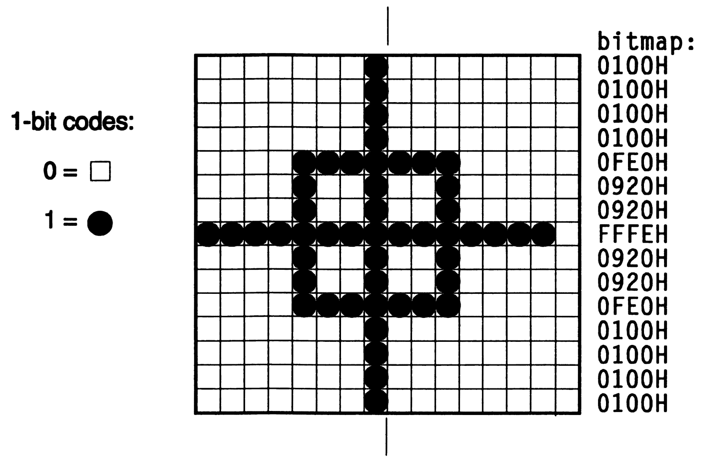


Figure 4.1 *One-Bit-per-Pixel Raw Image Bitmap*

## 4.0.1 Raw Image Data

The simplest possible image data encoding is a bare list of pixel attributes. This encoding, called the *raw image data*, is often all that is required by a graphics application. For example, the two-color bitmap in Figure 4.1 is encoded as raw image data.

The image in Figure 4.1 consists entirely of pixels set to a single color or attribute, represented by black dots in the illustration. Therefore each screen pixel can be encoded in a single memory bit. If the bit is 0, then the screen pixel is left in the background state. If the memory bit is 1, then the pixel is set to the single supported attribute. The resulting bitmap is encoded as a bit-per-pixel format.

Often a graphics application must encode more than one attribute per screen pixel. For example, Figure 4.2 is a representation that uses 3 attributes of the image in Figure 4.1, in addition to the background.

In Figure 4.2 each screen pixel can be in one of four attributes: background, light gray, dark gray, or black. In order to represent these four states it is necessary to assign a 2-bit field for each screen pixel. The four bit combinations that correspond to the attribute options are shown on the left side of Figure 4.2. At the bottom of Figure 4.2 is a map of one of the pixel rows, with the corresponding binary codes for each pixel, as well as the hexadecimal digits of the bitmap.

Comparing Figures 4.1 and 4.2 we see that as the number of attributes per pixel increases, the memory storage devoted to each pixel also grows. In Figure 4.1 a single bit encodes the two possible attributes that can be assigned to each pixel, while in Figure 4.2 a 2-bit field is necessary to represent the four possible pixel attributes. By the same token, if each pixel can be represented in one of 256 attributes (or colors), the encoding requires an 8-bit field to represent each pixel.

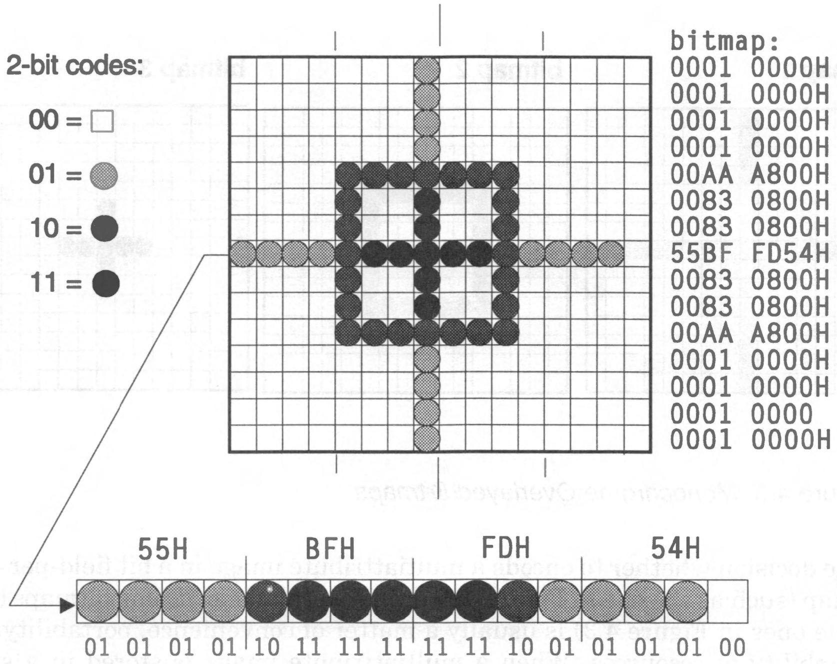


Figure 4.2 *Two-Bit-per Pixel Raw Image Bitmap*

The designer of a graphics application must often decide whether to use a customized format, including only the data strictly necessary for the display routine, or to represent the image in one of the more or less standard formats recognized by other graphics applications. The basis for this decision is usually one of image portability. A stand-alone program, which has no need to communicate graphics data to other applications, can often profit from a raw data format whenever it is convenient. On the other hand, an application that must exchange image data with other graphics programs benefits from adopting one of the existing image data formats described later in this chapter.

### 4.0.2 Monochrome and Color Bitmaps

The term monochrome means "of one color," although in computer jargon it is often interpreted as black-and-white. This assumption is not always true in bit-mapped graphics, since a monochrome bitmap can be displayed in any available color or attribute. Furthermore, it is possible to combine several monochrome bitmaps to form a multicolor image on the screen. In Figure 4.3 the image of Figure 4.2 has been separated into three monochrome bitmaps. If the software interprets bitmap 1 to be displayed in light gray pixels, bitmap 2 to be displayed in dark gray pixels, and bitmap 3 to be displayed in black pixels, the resulting overlayed image would be identical to the one in Figure 4.2.

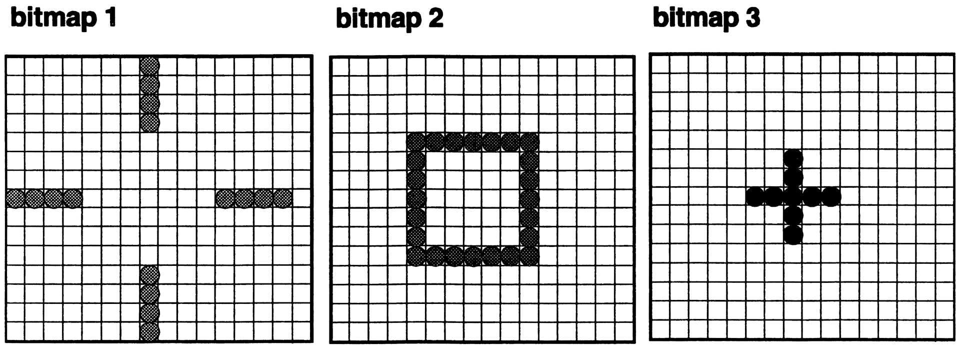


Figure 4.3 *Monochrome Overlayed Bitmaps*

The decision whether to encode a multiattribute image in a bit field-per-pixel bitmap (such as the one in Figure 4.2) or in several monochrome bitmaps (such as the ones in Figure 4.3) is usually a matter of convenience, portability, and availability of resources. When a multiattribute image is stored in a single bitmap, the result is a more compact image file and a faster display operation. On the other hand, several monochrome bitmaps can be easier to generate by means of a drawing program. For example, Figure 4.4 shows the three bitmaps used to display a color image of a target rifle. One bitmap encodes the pixels to be displayed with a black attribute, and the second bitmap encodes the displayed pixels in a brown attribute, representing the rifle's wood stock. The third bitmap encodes the highlights, that is, the pixels to be displayed with a bright white attribute.

The result of overlaying the three bitmaps in Figure 4.4 is a colored image. The advantages are compactness of encoding and ease of image manipulation. In animated programs these considerations are often very important.

### 4.0.3 Image Data Compression

Bit-mapped image data takes up considerable memory space. For example, the raw image data for a full screen, in an XGA or SuperVGA mode of 1024-by-768 pixels resolution in 256 colors, requires approximately 768K. This exceeds the user memory space available in an MS-DOS machine. Several data compression schemes have been devised to reduce the memory space required for storing pixel-coded images. However, image data compression is achieved at a price: the additional processing time required for packing and unpacking the image data. In animated applications performance is often such a critical factor that this overhead is an important consideration in adopting a compressed data format.

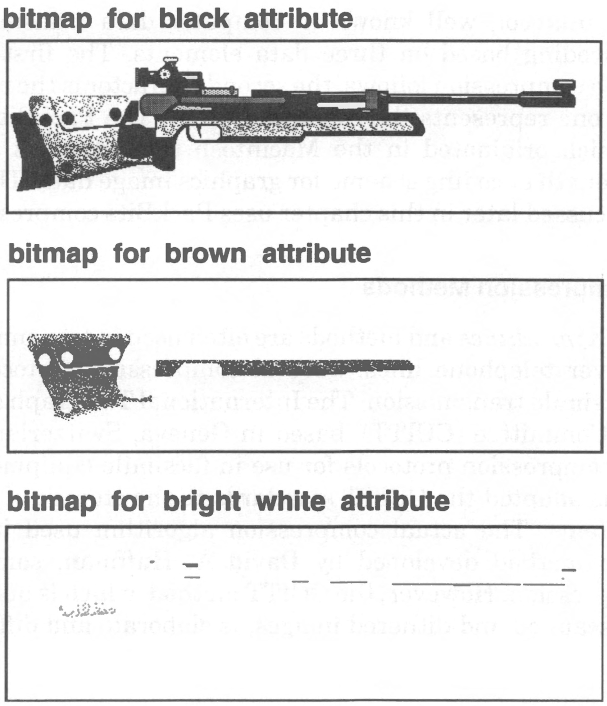

bitmap for black attribute
bitmap for brown attribute
bitmap for bright white attribute

Figure 4.4 *Overlayed Bitmaps for a Color Image*

Many of the compression methods used for alphanumeric data are not adaptable for image data. In the first place, all of the irreversible techniques used in character data compaction cannot be used for graphics images, since image data must be restored integrally. The same applies to the semantic-dependent methods developed for text compression. On the other hand, some general principles of data compression are applicable to graphics encoding schemes and can be used to compress pixel data. The following compression methods are applicable to graphics image data.

## Run-Length Encoding

The *run-length encoding* method is based on the suppression of repeated character codes. It is based on the principle that if a character is repeated three or more times, then the data string can be more compactly represented in coded form. Run-length encoding is a simple and efficient graphics data compression scheme based on the assumption that image data often contains entire areas of repeated pixel values. Notice that approximately two-thirds of the bitmaps shown in Figures 4.2 and 4.3 consist of NULL pixels (white background color). Even the nonbackground areas of the image contain strings with the same attribute. In this case a simple compression scheme could be used to pack the data in the white, black, dark gray, and light gray areas so as to save considerable image storage space.

The Kermit protocol, well known in computer data transmission, uses a run-length encoding based on three data elements. The first code element indicates that a compression follows, the second character is the repetition code, and the third one represents the repetition count. The PackBits compression algorithm, which originated in the Macintosh computers, is an even more efficient run-length encoding scheme for graphics image data. The TIFF image file format discussed later in this chapter uses PackBits compression encoding.

## Facsimile Compression Methods

*Facsimile* (FAX) *machines* and methods are often used in transmitting graphics image data over telephone lines. Several compression protocols have been devised for facsimile transmission. The International Telegraph and Telephone Consultative Committee (CCITT), based in Geneva, Switzerland, has standardized data compression protocols for use in facsimile equipment. The TIFF convention has adapted the CCITT standards to the storage of image data in computer systems. The actual compression algorithm used in CCITT is a variation of a method developed by David A. Huffman, sometimes called Huffman compression. However, the CCITT method, which is quite efficient for monochrome scanned and dithered images, is elaborate and difficult to implement.

## LZW Compression

LZW is a compression technique suited to color image data. The method is named after Abraham Lempel, Jabob Ziv, and Terry Welch. The algorithm, also known as Ziv-Lempel compression, was first published in 1977 in an article by Ziv and Lempel in the *IEEE Transactions on Information Theory*. The compression technique was refined by Welch in an article titled "A Technique for High-Performance Data Compression" that appeared in *Computer*, in 1984. LZW compression is based on converting raw data into a reversible encoding in which the data repetitions are tokenized and stored in compressed form. LZW compression is used in many popular data and image compression programs, including the Compuserve GIF image data encoding format and in some versions of the TIFF standard. LZW compression has been patented by Unisys Corporation. Therefore, its commercial use requires a license from the patent holders. The following statement is inserted at the request of Unisys Corporation:

> "The LZW data compression algorithm is said to be covered by U.S. Patent 4,558,302 (the "Welch Patent"). The Welch Patent is owned by Unisys Corporation. Unisys has a significant number of licensees of the patent and is committed to licensing the Welch Patent on reasonable non-discriminatory terms and conditions. For further information, contact Unisys Welch Licensing Department, P.O. Box 500, Blue Bell, PA 19424, M/S C1SW19."

The LZW algorithm is explained in detail in our book *Graphics Programming Solutions* (McGraw-Hill, 1993).

### 4.0.4 Encoders and Decoders

An encoder is a program or routine used to convert raw image data into a standard format. We speak of a GIF encoder as a program or routine used to store a graphics image in a file structured in the GIF format. A decoder program or routine performs the reverse operation; that is, it reproduces the graphics image or the raw data from the information stored in an encoded image file. In the more conventional sense, a GIF decoder displays on the screen an image file stored in the Compuserve GIF format. Therefore the fundamental tool kit for operating with a given image data format consists of encoder and decoder code. With some compressed image formats the processing required in encoders and decoders can be quite elaborate.

## 4.1 The GIF Format

The *Graphics Interchange Format* (GIF) originated in the Compuserve computer information service. The first description of the GIF protocol, which appeared on the Compuserve Picture Support Forum on May 28, 1987, was identified with the code letters GIF87a, while the current version is labeled GIF89a. GIF is the only graphics image storage format in use today that is not associated with any software company. Although the GIF standard is copyrighted, Compuserve grants royalty-free adoption rights to anyone wishing to use it. According to Compuserve, software developers are free to use the GIF encodings by accepting the terms of the Compuserve licensing agreement, which basically states that all changes to the standard must be made by the copyright holders and that the software utilizing GIF must acknowledge Compuserve's ownership. The agreement can be obtained from the Compuserve Graphics Technology Department or in the graphics forums.

GIF was conceived as a compact and efficient storage and transmission format for computer imagery. The GIF87a specification supports multiple images with a maximum of 16,000-by-16,000 pixels resolution in 256 colors. This format is suited to the maximum resolution available today in SuperVGA and XGA systems.

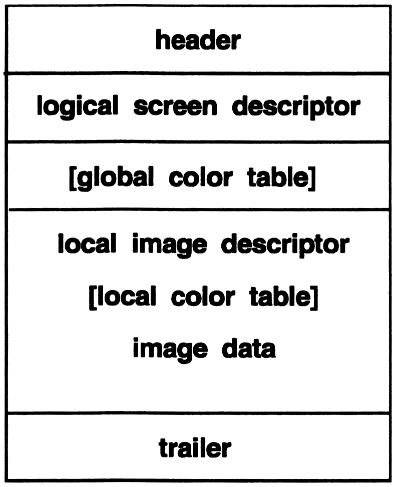
The advantages of the GIF standard are related to its being compact, powerful, portable, and, presumably, public. There is an extensive collection of public domain images in GIF format which can be found in the Compuserve graphics forums and in many bulletin board services. The major disadvantage of the GIF standard is that many commercial programs do not support it. Users of popular graphics programs soon discover that GIF is not included in the relatively extensive catalog of file formats which the application imports and exports. This limitation can often be solved by means of a conversion utility that translates a format recognized by the particular application into a GIF encoding. Several of these format conversion utilities are available on the Compuserve graphics forums.

The main sources of information about the GIF standard are the graphics forums on the Compuserve Information Service. The specifications of GIF89a are available in the file GIF89A.DOC found in library number 14 of the Compuserve Graphics Support forum. Image files in the GIF format are plentiful on the Compuserve Graphics Support libraries as well as in many bulletin board services.

### 4.1.1 GIF File Structure

The two versions of the GIF standard at the time of this writing are labeled GIF87a and GIF89a. Version 89a, an extension of version 87a, adds several features to the original GIF protocol, such as the display of text messages, comments, and application and graphics control data. The detailed description of the GIF protocol is found in the file GIF89A.DOC mentioned in the previous section. The following description is limited to the features common to both the GIF87a and GIF89a specifications.

The GIF87a format is defined as a series of blocks and sub blocks containing the data necessary for the storage and reproduction of a computer graphics image. A GIF data stream contains the data stored in these blocks and sub blocks in the order defined by the GIF protocol. The first block in the data stream is the *header* and the last one is the *trailer*. Image data and other information are encoded between the header and trailer blocks. These can include a logical screen descriptor block and a global color table, as well as one or more local image descriptors, local color tables, and compressed image data. The GIF89a protocol allows graphics control and rendering blocks, plain text blocks, and an application data block. Figure 4.5 shows the elements of the GIF87a data stream.



Note: items in braces are optional

Figure 4.5 *The GIF Data Stream*

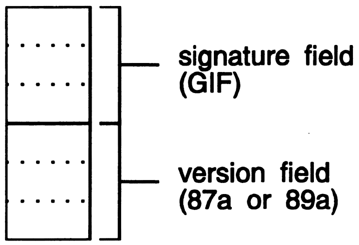


Figure 4.6 *The GIF Header*

## GIF Header

The first item in the GIF data stream is the header. It consists of six ASCII characters. The first three characters, called the signature, are the letters "GIF." The following three characters encode the GIF version number. The value "87a" in this field refers to the version of the GIF protocol approved in May 1987, while the value "89a" refers to the GIF version dated July 1989. Figure 4.6 shows the elements of the GIF header.

One header must be present in each GIF data stream. A GIF encoder must initialize all six characters in the GIF header. The version number field should correspond to the earliest GIF version that defines all the blocks in the actual data stream. In other words, a GIF file that uses only the elements of the GIF87a protocol should contain the characters 87a in the version field of the GIF header, even if the file was created after the implementation of the GIF89a protocol. The GIF decoder uses the information in the header block to certify that the file is encoded in the GIF format and to determine version compatibility.

## GIF Logical Screen Descriptor

The block immediately following the header is called the *logical screen descriptor*. This block contains the information about the display device or mode compatible with the image. One logical screen descriptor block must be present in each GIF data stream. Figure 4.7 shows the elements of the logical screen descriptor block.

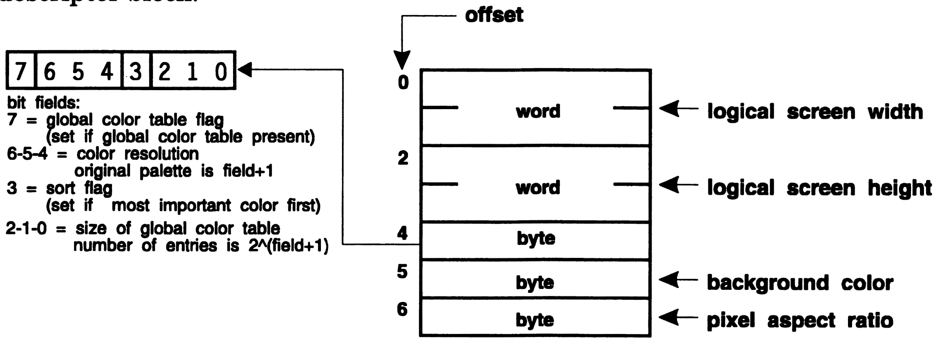

offset
7 6 5 4 3 2 1 0
bit fields:
7 = global color table flag
(set if global color table present)
6-5-4 = color resolution
original palette is field+1
3 = sort flag
(set if most important color first)
2-1-0 = size of global color table
number of entries is 2^(field+1)
0
2
4
5
6
word
word
byte
byte
byte
logical screen width
logical screen height
background color
pixel aspect ratio

Figure 4.7 *GIF Logical Screen Descriptor*

The fields of the GIF logical screen descriptor are formatted as follows:

1. The words at offset 0 and 2, labeled *logical screen width* and *logical screen height,* encode the pixel dimensions of the logical screen to be used by the display device. In IBM microcomputers this value usually coincides with the selected display mode.
2. The byte at offset 4 is divided into four bit fields. Bit 7, labeled the *global color table flag*, serves to indicate if a global color table is present in the data stream that follows. The global color table is discussed later in this section. Bits 6, 5, and 4 are the *color resolution field*. This value represents the number of palette bits for the selected mode, plus one. For example, a 16-color VGA palette (4 bits encoding) is represented by the bit value 011 (decimal 3). Bit 3, labeled the *sort flag*, is used to signal that the global color table (if present) is sorted starting with the most important colors. This information can be used by the software if the display device has fewer colors available than those used in the image. Finally, the field formed by bits 2, 1, and 0 determines the *size of the global color table* (if one is present). The value is encoded as a power of 2, diminished by 1. Therefore, to restore the original exponent it is necessary to add 1 to the value encoded in the bit field. For example, a bit value of 011 (3 decimal) corresponds to a global color table representing $2^4$, or 16 colors. Notice that this value corresponds to the number of colors in the global color table, not to its byte length (discussed later in this section). The maximum representable value in a 3-bit field is 7, which limits the number of colors in the global color table to $2^8$, or 256 colors.
3. The field at offset 5, labeled *background color* in Figure 4.7, is used to represent the color of those pixels located outside of the defined image or images. The value is an offset into the global color table.
4. The field at offset 6, labeled the *pixel aspect ratio* in Figure 4.7, is used to compensate for nonproportional $x$ and $y$ dimensions of the display device. This field is set to zero for systems with a symmetrical pixel density, such as the most widely used modes in VGA and XGA systems.

## GIF Global Color Table

The *global color table* is an optional GIF block used to encode a general color palette for displaying images in data streams without a local color table. The global color table serves as a default palette for the entire stream. Recall that the GIF data stream can contain multiple images. The presence of a global color table and its size is determined from the data furnished in the logical screen descriptor block (see Figure 4.5). Only one global color table is present in the data stream. Figure 4.8 shows the structure of a global color table.

The entries in the global color table consist of values for the red, green, and blue palette registers. Each component color takes up one byte in the table; therefore each palette color consists of three bytes in the global color table. The number of entries in the global color table is determined by reading bits 0, 1, and 2 of the global color size field in the logical screen descriptor block (see Figure 4.5). The byte length of the table is three times the number of entries. The maximum number of palette colors is 256. In this case the global color table takes up 768 bytes (see Figure 4.8).

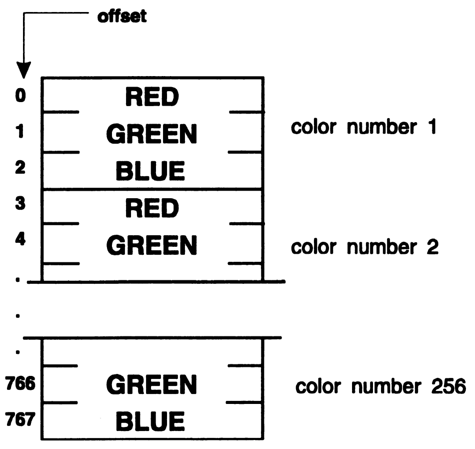


Figure 4.8 *GIF Global Color Table*

## GIF Image Descriptor

Each image in the GIF data stream is defined by an image descriptor, an optional local color table, and one or more blocks of compressed image data. The *image descriptor* block contains the information for decoding and displaying the image. Figure 4.9 shows the elements of the image descriptor block.

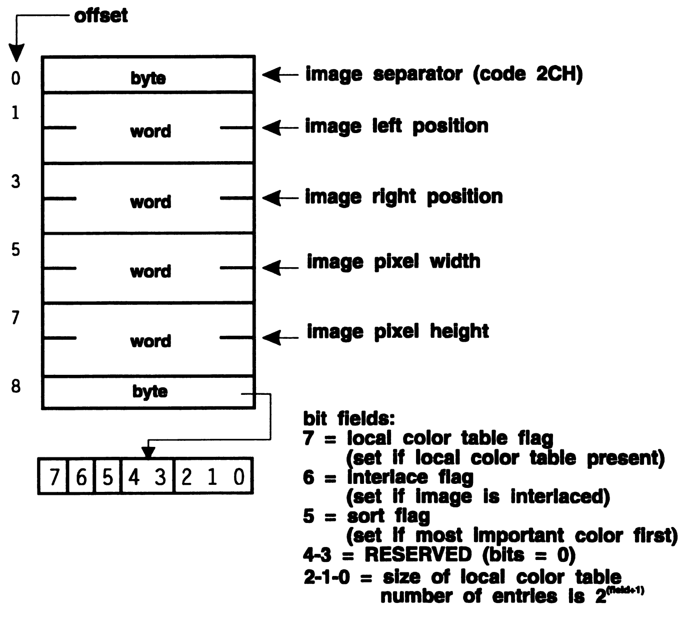

offset
0
1
3
5
7
8
byte
word
word
word
word
byte
image separator (code 2CH)
image left position
image right position
image pixel width
image pixel height
7 6 5 4 3 2 1 0
bit fields:
7 = local color table flag
(set if local color table present)
6 = interlace flag
(set if image is interlaced)
5 = sort flag
(set if most important color first)
4-3 = RESERVED (bits = 0)
2-1-0 = size of local color table
number of entries is 2^(field+1)

Figure 4.9 *GIF Image Descriptor*

The fields of the GIF image descriptor are formatted as follows:

1. The byte at offset 0, labeled image separator in Figure 4.9, must be the code 2CH.
2. The words at offset 1 and 3, labeled image left position and image right position, respectively (see Figure 4.9), encode the screen column and row coordinates of the image's top-left corner. This location is an offset within the logical screen defined in the logical screen descriptor block (see Figure 4.5).
3. The words at offset 5 and 7, labeled image pixel width and image pixel height, respectively (see Figure 4.9), encode the size of the image, measured in screen pixels.
4. The byte at offset 8 in Figure 4.9 is divided into five bit fields. Bit 7, labeled the local color table flag, serves to indicate if a local color table follows the image descriptor block. If a local color table is present in the data stream, it is used for displaying the image represented in the corresponding descriptor block. Bit 6, labeled the interlace flag, encodes if the image is interlaced, that is, if its rows are not arranged in consecutive order. In the PC interlaced images are used in some CGA and EGA display modes, but not in the proprietary VGA and XGA modes. Bit 5, labeled the sort flag, is used to signal that the local color table (if present) is sorted starting with the most important colors. This information is used by the software if the display device has fewer available colors than those in the table. The field formed by bits 2, 1, and 0 determines the size of the local color table (if one is present). The value is encoded as a power of 2, diminished by 1. Therefore, to restore the original exponent, it is necessary to add 1 to the value encoded in the bit field. For example, a bit value of 011 (3 decimal) corresponds to a global color table representing $2^4$, or 16 colors. Notice that this value corresponds to the number of colors in the local color table, not to its byte length (refer to the previous discussion about the global color table).

## GIF Local Color Table

The local color table is an optional GIF block that encodes the color palette used in displaying the image corresponding to the preceding image descriptor block. If no local color table is furnished, the image is displayed using the values in the global color table. If neither table is present, it is displayed using the current setting of the DAC registers. The GIF data stream can contain multiple images, with each one having its own local color table. The structure of the local color table is identical to the one described for the global color table (see Figure 4.8).

## GIF Compressed Image Data

The image itself follows the local color table, if one is furnished, or the image descriptor block if the data stream does not include a local color table. The GIF standard sets no limit to the number of images contained in the data stream. Image data is divided into sub blocks, with each sub block having at the most 255 bytes. The data values in the image are offsets into the current color palette. For example, if the palette is set to standard IRGB code, a pixel value of 1100B (decimal 12) corresponds to the twelfth palette entry, which, in this case, encodes the LUT register settings for bright red. Preceding the image data blocks is a byte value that holds the code size used for the LZW compression of the image data in the stream. This data item normally matches the number of bits used to encode the pixel color. For example, an image intended for VGA mode 18, in 16 colors, has an LZW code size of four, while an image for VGA mode 19, in 256 colors, has an LZW code size of eight. Figure 4.10 shows the format of the GIF data blocks.

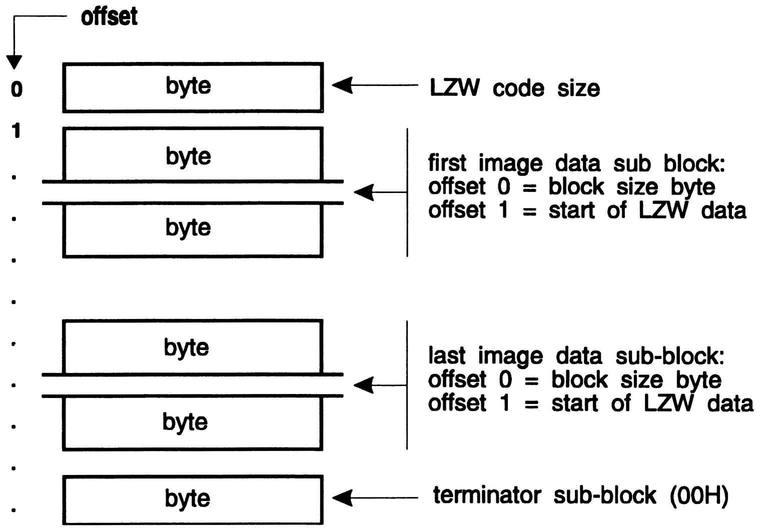

offset
0
1
byte
LZW code size
first image data sub block:
offset 0 = block size byte
offset 1 = start of LZW data
last image data sub-block:
offset 0 = block size byte
offset 1 = start of LZW data
terminator sub-block (00H)

Figure 4.10 *GIF Image Data Blocks*

The image data sub blocks contain the image data in compressed form. The LZW compression algorithm used in the GIF protocol is explained in our book *Graphics Programming Solutions* (McGraw-Hill, 1993). Each data sub block starts with a block-size byte, which encodes the byte length of the data stored in the rest of the sub block. The count, which does not include the count byte itself, can be in the range 0 to 255. The compressed data stream ends with a sub block with a zero byte count (see Figure 4.10).

## GIF Trailer

The simplest GIF block is named the *trailer*. This block consists of a single byte containing the GIF special code 3BH. Every GIF data stream must end with the trailer block. The GIF trailer is shown in Figure 4.11.

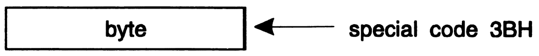


Figure 4.11 *GIF Trailer*

### GIF89a Extensions

GIF version 89a contains several features that are not present in version 87a. These include the following new blocks:

1. A *graphics control extension* refers to a graphics rendering block, also a new feature introduced in version 89a. The graphics control extension contains information on displaying the rendering block. This information includes instructions about disposing of the currently displayed image, handling the background color, action on user input, time delay during the display operation, and image transparency.
2. The *graphics rendering blocks* can be an image descriptor block, as described for GIF version 87a, or a new *plain text extension*. The plain text extension contains ASCII data to be displayed in a coarse grid of character cells determined in the block. Also in the plain text block are the foreground and background colors, the coordinates of the start position, and the text message itself.
3. The *applications extension* is an extension block in GIF version 89a that contains application-specific information. The block includes an 8-byte application identifier field intended for an ASCII string that identifies the particular piece of software. A 3-byte authentication code follows the identifier. Application data follows the authentication code field.

## 4.2 The TIFF Format

The *Tag Image File Format* (TIFF) was developed by Aldus Corporation with the support of several other companies, including Hewlett-Packard and Microsoft. The standard intends to provide a flexible file storage format for raster images. Its origin is related to scanner hardware and software for microcomputers. The first version of TIFF was published in the fall of 1986. The present update, designated as TIFF Revision 6.0, was released in June 1992. TIFF is a nonproprietary standard which can be used without license or previous royalty agreement. Technical information about TIFF can be obtained from the Aldus Developer's Desk at Aldus Corporation, Seattle, Washington, or from the Aldus forum on Compuserve (GO ALDSVC).

The purpose of the TIFF standard is to provide an image storage convention with maximum flexibility and portability. TIFF is not intended for any particular computer, operating system, or application program. Consistent with this idea, the files in TIFF format have no version number or other update identification code. A typical TIFF reader searches for the necessary data and ignores all other information contained in the file. The format supports both the Intel and the Motorola data ordering schemes, but hardware-specific features are not documented in the TIFF file. The mode, resolution, or color range used in displaying a TIFF file is left entirely to the software.

The TIFF standard supports monochrome, grayscale, and color images of various specifications. The original TIFF documents classified the various image types into four classes. Class B was used for binary (black-and-white) images, class G for grayscale images, class P for palette color images (8-bits-per-pixel color), and class R for full-color images (24-bits-per-pixel color). A TIFF application need not provide support for all TIFF image types. For example, a VGA TIFF reader could exclude class R images since the system's maximum color range is 8 bits-per-pixel (256 colors). By the same token, a routine or application that reads monochrome scanned images could limit its support to the class B category. The image class designation by letter codes was dropped in TIFF revision 6.0; however, the image classification into bilevel, grayscale, RGB, and palette types was preserved.

TIFF originally supported uncompressed images as well as compressed data according to several compression schemes, namely, PackBits, CCITT, and LZW. LZW compression support was dropped in TIFF version 6.0, because the compression algorithm is patented by Unysis Corporation. Notice that in the TIFF standard, compression methods are usually associated with the particular file classes mentioned in the preceding paragraph.

### 4.2.1 TIFF File Structure

The TIFF standard is an image file protocol. A file in the TIFF format is divided into three areas: the header, the image file directory, and the image data.

The notion of *tags* is the feature that identifies files in the TIFF format. A TIFF tag is a word integer that serves to identify the file structure that follows. For example, the tag value 103H indicates that the structure that follows contains data compression information. TIFF file processing software can search for this tag in order to determine which, if any, compression scheme was used in encoding the image data. TIFF tags are discussed in greater detail later in this section.

#### TIFF Header

An image file in TIFF format must start with an 8-byte block called the header. Figure 4.12 shows the structure of the TIFF image file header.

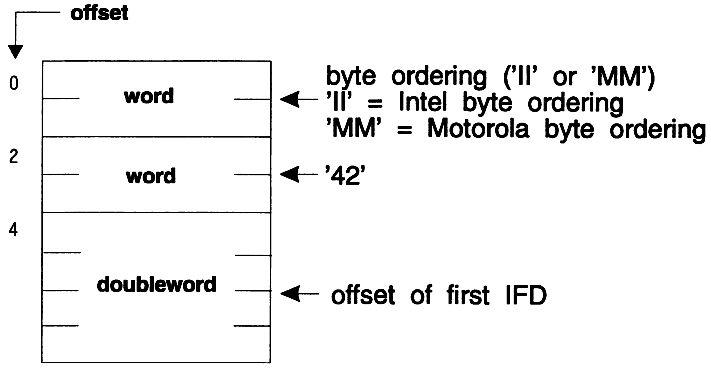

offset
0
2
4
word
word
doubleword
byte ordering ('II' or 'MM')
'II' = Intel byte ordering
'MM' = Motorola byte ordering
'42'
offset of first IFD

Figure 4.12 *TIFF File Header*

The word at offset 0 of the TIFF file header consists of the ASCII characters 'II' or 'MM'. The 'II' code identifies a file in the Intel byte ordering scheme; that is, word and doubleword entries appear with the least significant byte in the lowest numbered memory address. This data ordering format is sometimes known as the "little-endian" scheme. The 'MM' code identifies a file in the Motorola byte ordering order, that is, with the least significant byte of word and doubleword entries in the highest numbered memory address. This format is known as the "big-endian" scheme. The ASCII number '42' found at the word at offset 2 of the header serves to further identify a file in TIFF format. The numbers themselves have no documented significance. The ASCII code '42' has sometimes been called the TIFF version number, although it is not described as such in the standard. The doubleword at offset 4 of the header block contains the offset, in the TIFF file, of the first *image file directory* (IFD).

The file header block is the only TIFF file structure that must be located at a predetermined offset from the start of the file. The remaining structures can be located anywhere in the TIFF file. TIFF file processing code reads the data in the header block to certify that the file is in TIFF format and to make decisions regarding the data ordering scheme. A sophisticated application could be capable of making adjustments in order to read data both in Intel and in Motorola orders, while another one could require data in a specific format.

## TIFF Image File Directory (IFD)

Once the code determines that the file is in TIFF format and that it is encoded in a valid ordering scheme, it uses the doubleword at offset 4 of the header (see Figure 4.12) to determine the location of the first image file directory (IFD). Notice that a TIFF file can contain more than one image. If so, each image in the file is associated with its own IFD. However, by far the more common situation is that a TIFF file contains a single image. This assumption is made in the code and examples for manipulating TIFF files. The structure of the IFD is shown in Figure 4.13.

Observe that the offset values in the left-most column of Figure 4.13 (labeled "local offset") refer to offsets within the IFD block because the IFD itself can be located anywhere within the TIFF file. The word at local offset 0 of the IFD is a count of the number of directory entries. Recall that the number of directory entries is unlimited in the TIFF standard. The last directory entry is followed by a doubleword field which contains the offset of the next IFD, if one exists. If not, this doubleword contains the value 0000H. Each entry in the IFD takes up 12 bytes. The structure of each IFD entry is shown in Figure 4.14.

The tag code is located at local offset 0 in the directory entry field. TIFF requires that the entry fields be sorted by increasing order of the tag codes; therefore, a lower numbered tag code always precedes a higher numbered one. This simplifies searching for a particular tag code since the search terminates when one with a higher numbered tag is encountered. The type code is located at local offset 2 within the directory entry field.

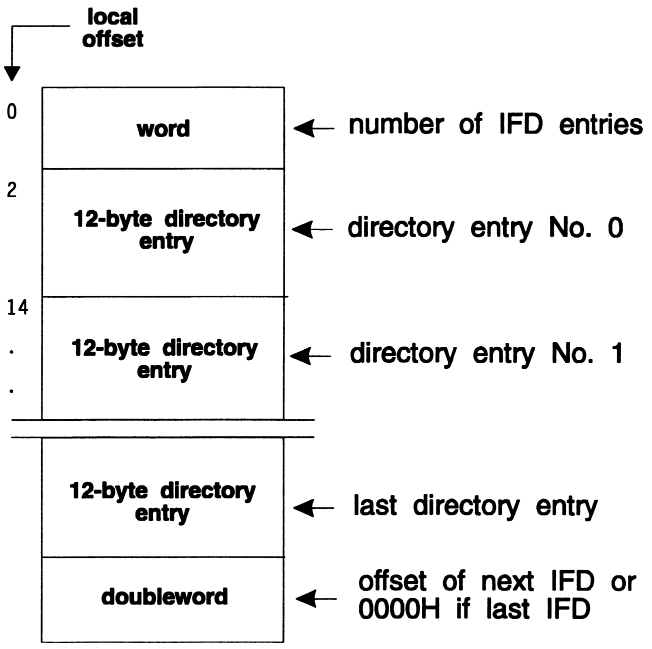


Figure 4.13 *TIFF Image File Directory*

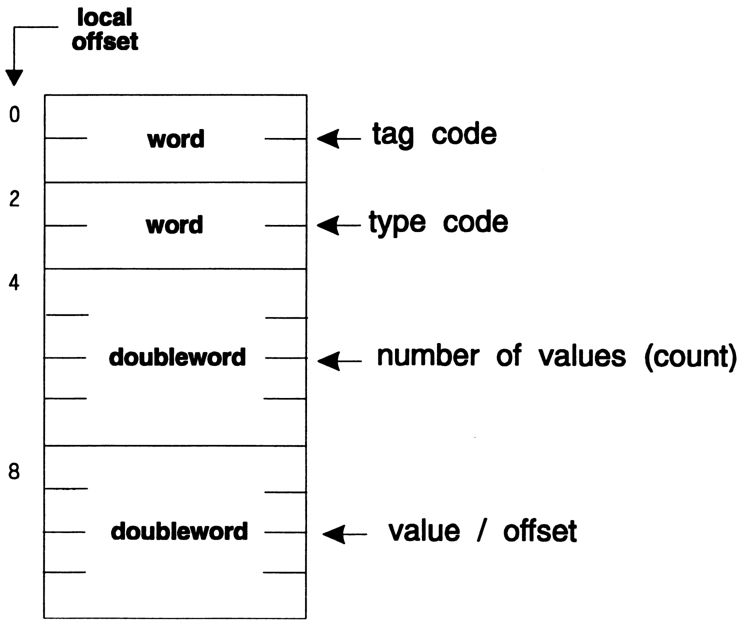


Figure 4.14 *TIFF Directory Entry*

Table 4.1 shows the type code values according to TIFF version 6.0. Code numbers six and higher were introduced in Version 6.0 and are not documented in previous versions of the standard.

Table 4.1 *TIFF Version 6.0 Field Type Codes*

| TYPE CODE | STORAGE UNIT | FIELD CONTENTS |
|---|---|---|
| 1 | byte | 8-bit unsigned integer |
| 2 | ASCII character | offset of ASCII string terminated in NULL byte |
| 3 | word | 16-bit unsigned integer |
| 4 | doubleword | 32-bit unsigned integer |
| 5 | quadword | Rational number. The first doubleword is the numerator of a fraction and the last doubleword is the denominator |
| 6 | byte | 8-bit signed integer |
| 7 | byte | Undefined. Can be used at will by the software |
| 8 | word | 16-bit signed integer in 2's complement form |
| 9 | doubleword | 32-bit signed integer in 2's complement form |
| 10 | quadword | Rational number. The first doubleword is the signed numerator of a fraction and the last doubleword is the signed denominator |
| 11 | doubleword | Single precision floating-point number in IEEE format |
| 12 | quadword | Double precision floating-point number in IEEE format |

The *count* field is a doubleword at offset 4 of the directory entry. This field, which was named the *length* field in previous versions of TIFF, encodes the number of data repetitions in the current directory entry. Notice that this value does not encode the number of bytes, but the number of data units. For example, if the field type code is 3 (word unit) then the count field represents the number of data words of information that are associated with the entry.

The *value / offset* field is designated in this manner because it contains either a direct value or an offset into the TIFF file. The general rule is that if the encoded data fits into a doubleword storage (4 bytes), then the data is entered directly in the doubleword at local offset eight of the directory entry (see Figure 4.14). This design saves coding space and simplifies processing. However, some TIFF tags, such as the StripOffset tag mentioned later in this section, always contain offset data in this field. The software determines if the data in the value/offset field is either a value or an offset by means of the tag, the field type code, and the data item count.

If the tag contains either a value or an offset, the program must first examine the field type codes (see Table 4.1). In this case data corresponding to field type codes 1, 3, 4, 5, 6, 7, 8, 9, and 11, is contained in a doubleword storage unit and is therefore entered as values. By the same token, field types 2, 5, 10, and 12 encode an offset in the value/offset field of the directory entry. Once it is determined that an individual data item fits in the 4 bytes allocated to the value/offset field, then the software must examine the number of values associated with the directory entry. If the total number of values exceeds the allocated space (4 bytes), then the value/offset field contains an offset. In this case the type code and the count fields are multiplied in order to determine the number of items supplied.

## 4.2.2 TIFF Tags for Bilevel Images

Over 50 tags have been defined in the TIFF standard; however, only a handful are used in most TIFF images. A complete description of all the TIFF tags is found in TIFF Revision 6.0 specification available, at no charge, from Aldus Corporation. The TIFF tags mentioned in the following discussion are those that would be commonly found in monochrome (bilevel in TIFF terminology) scanned images.

### OldSubFileType (tag code 00FFH)

This tag, originally called the SubFileType, has been replaced by the NewSubFileType tag; however, many older TIFF programs still use this tag. The tag provides information about the bitmap associated with the IFD. The tag can take the following values:

Value = 1 indicates that the image is in full-resolution format.

Value = 2 indicates the image data is in reduced-resolution format.

Value = 3 indicates that the image data is a single page of a multipage image.

### NewSubFileType (tag code 00FEH)

This tag, which replaces OldSubFileType, describes the kind of data in the IFD. The tag is made up of a doubleword integer with the following significant bits:

Bit 0 is set if the image is a reduced-resolution version of another image.

Bit 1 is set if the image is a single page of a multipage image.

Bit 2 is set if the image is a transparency mask (see the PhotometricInterpretation tag later in this section.)

### ImageWidth (tag code 0100H)

This tag encodes the number of pixel columns in the image.

### ImageLength (tag code 0101H)

This tag encodes the number of pixel rows in the image.

### BitsPerSample (tag code 0102H)

This tag encodes the number of bits required to represent each pixel sample. The value of this tag is one for bilevel images, four for 16-color palette images, and eight for 256-color palette images. In IBM video graphics systems the number of bits per sample is usually the same as the number of bits per pixel color. Regarding images encoded in RGB format (as used in some Macintosh systems and in the XGA Direct Color mode), the number of bits per sample refers to each individual color. In this case the SamplesPerPixel tag encodes the number of pixel colors (three colors in RGB encoding), and the BitsPerSample tag the number of bits assigned to each color. For example, if six bits are assigned to the red sample, eight bits to the green, and six bits to the blue, the total number of bits per pixel would be 20.

### Compression (tag code 0103H)

This tag encodes the compression scheme used in the image data. The tag can take the following values:

Value = 1 indicates that the image data is not compressed. Pixel information is packed at the byte level, as tightly as possible. Uncompressed data has a disadvantage over compressed data in that it takes up more memory space. On the other hand, it has an advantage in that it can be manipulated faster by the display routines.

Value = 2 indicates that image data is compressed according to CCITT Group 3 (Modified Huffman) run-length encoding.

Value = 32,773 (8005H) indicates the data is compressed according to the PackBits scheme described in detail later in this section.

### PhotometricInterpretation (tag code 0106H)

This tag describes how to interpret the color encoding in the bitmap. The tag can take the following values:

Value = 0 is used in bilevel and grayscale images to indicate that a bit value of 0 represents the white color.

Value = 1 is used in bilevel and grayscale images to indicate that a bit value of 0 represents the black color.

Value = 2 is used to indicate an encoding in RGB format.

Value = 3 is used to indicate palette color format. In this case a ColorMap tag must be included to hold the LUT values.

Value = 4 indicates that the image is a transparency mask used to define an irregularly shaped region of another image.

### Threshholding (tag code 0107H)

This tag describes the technique used for representing the gray scale in a black-and-white image. The tag can have the following values:

Value = 1 indicates that the image contains no dithering or halftoning. Bilevel images use this value.

Value = 2 indicates that the image has been dithered or halftoned.

Value = 3 indicates that a randomized process, such as the error diffusion algorithm, has been applied to the image data.

### StripsOffset (tag code 0111H)

This tag provides the information necessary for the software to locate the image data within the TIFF file. By definition, the value in this tag is always an offset from the beginning of the TIFF file. The structure of the TIFF image data, as well as the use of this tag, is discussed in Section 4.2.3.

### SamplesPerPixel (tag code 0115H)

This tag encodes the number of color components for each screen pixel. The value of this tag is one for bilevel, grayscale, and palette color images, and three for images in RGB format.

### RowsPerStrip (tag code 0116H)

This tag determines the number of rows in each strip. Image encoding in the TIFF standard is discussed in Section 4.2.3.

### StripByteCounts (tag code 0117H)

This tag determines the number of bytes in each strip, after compression. Image encoding in the TIFF standard is discussed in Section 4.2.3.

### XResolution (tag code 011AH)

This tag provides information about the *x*-axis resolution at which the original image was created or scanned. The data is important to software that must reproduce the image exactly as it was originally produced. This is a critical factor in the reproduction of dithered images, which do not allow scaling.

### YResolution (tag code 011BH)

This tag provides information about the *y*-axis resolution at which the original image was created or scanned. See the text in the XResolution tab.

### PlanarConfiguration (tag code 011CH)

This tag provides information regarding the organization of color pixel data. It is relevant only for color images in RGB format (more than one sample per pixel). The tag can have the following values:

Value = 1 indicates that RGB data is stored in the order of the color components, that is, in a repeating sequence of red, green, and blue values. This organization is called the *chunky format* in TIFF documentation.

Value = 2 indicates that RGB data is stored by bit planes. That is, the red color components are stored first, followed by the green, and then by the blue. This organization is called the *planar format* in TIFF documentation.

### ResolutionUnit (tag code 128H)

This tag determines the unit of measurement used in the parameters contained in XResolution and YResolution tags. Many TIFF programs do not use this tag, but it is recommended by the standard. The tag can have the following values:

Value = 1 indicates no unit of resolution.

Value = 2 indicates inches.

Value = 3 indicates centimeters.

## 4.2.3 Locating TIFF Image Data

Although TIFF file processing software often ignores many tags and makes assumptions regarding others, one necessary manipulation in an image display operation is the locating and decoding of the image bitmap.

TIFF Image data can be located almost anywhere in the file. This is true of both uncompressed and compressed data. Furthermore, the TIFF standard allows dividing an image into several areas, called strips. The idea is to facilitate data input and output in machines limited to a 64K segment size. This is the case of Intel processors operating in MS-DOS or Windows systems. The data for each individual strip is represented by a separate tag.

When the image is divided into strips, three tags participate in locating the image data: RowsPerStrip, StripOffsets, and StripByteCounts. The first operation is for the software to calculate the number of strips into which the image data is divided. This value, which is not encoded in any particular tag, can be obtained from the number of values field of the StripOffsets tag (see Figure 4.14).

Locating the image data in a single strip image consists of adding the value in the StripOffsets tag to the start of the TIFF file. In this case the image size (in bytes) is obtained by reading the value in the ImageWidth tag (which is the number of pixels per row), dividing it by eight to determine the number of data bytes per pixel row, and multiplying this value by the number of pixel rows stored in the ImageLength tag.

If the image data consists of multiple strips, then each strip is handled separately by the software. In this case the number of bytes in each strip, after compression, is obtained from the corresponding entry in the StripByteCounts tag. The display routine obtains the number of pixel rows encoded in each strip from the value stored in the RowsPerStrip tag. However, if the total number of rows, as stored in the ImageLength tag, is not an exact multiple of the RowsPerStrip value, then the last strip could contain less rows than the value in the RowsPerStrip tag. TIFF software is expected to detect and handle this special case.

## 4.2.4 Processing TIFF Image Data

Once the start of the TIFF image data is located within the TIFF file, the code must determine if the data is stored in compressed or uncompressed format and proceed accordingly. This information is found in the Compression tag previously mentioned. In TIFF Version 5.0 the Compression tag could hold one of six values. Value number 1 corresponds to no compression, values 2, 3, and 4 correspond to three modes of CCITT compression, and value 5 corresponds to LZW compression; finally value 32,773 in the Compression tag indicates PackBits compression.

We mentioned that several of these compression schemes were dropped in Version 6.0 of the TIFF standard (see Section 4.2). In the present TIFF implementation, values 3, 4, and 5 for the Compression tag are no longer supported. Since there are substantial reasons to favor the LZW algorithm for the compression of color images (which was dropped in TIFF Version 6.0 because of patent rights considerations), we have limited the discussion on TIFF image decoding to the case of PackBits compression.

## TIFF PackBits Compression

The PackBits compression algorithm was originally developed on the Macintosh computer. The MacPaint program uses a version of PackBits compression for its image files. Macintosh users have available compression and decompression utilities for files in this format. The compression scheme is simple to implement and often offers satisfactory results with monochrome and scanned images.

PackBits, as implemented in TIFF, is a byte-level, simplified run-length compression scheme. The encoding is based on the value of the first byte of each compressed data unit, often designated as the *n* byte. The decompression logic can be described in the following steps:

```
STEP 1: If end-of-information code, then end decompression.
STEP 2: Read next source byte. Designate as $n$ ($n$ is an unsigned integer).
STEP 3: If $n$ is in the range 0 to 127 (inclusive), perform the following
        operations:
          a. Read the next $n+1$ bytes literally from the source file
             into the output stream.
          b. Go to STEP 1.
STEP 4: If $n$ is in the range 129 to 255 (inclusive), perform the following
        operations:
          a. Negate $n$ ($n = -n$).
          b. Copy the next byte $n+1$ times to the output stream.
          c. Go to STEP 1.
STEP 5: Goto STEP 1.
```

Notice that in the above description we assume that *n* is an unsigned integer. This convention, which facilitates coding in 80x86 assembly language, differs from other descriptions of the algorithm in which *n* is a signed value. Figure 4.15 is a flowchart of this decompression logic.

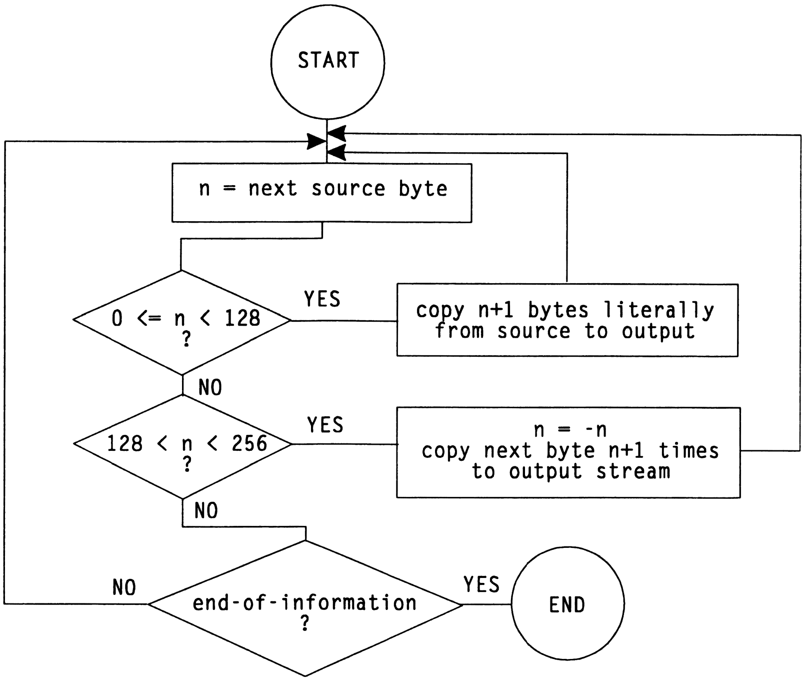


Figure 4.15 *TIFF PackBits Decompression*

Observe that in the TIFF implementation of PackBits no action is taken if $n = 128$. If $n = 0$, then one byte is copied literally from source to output. The maximum number of bytes in a compression run is 128. In addition, the TIFF implementation of PackBits compression adopted the following special rules:

1. Each pixel row is compressed separately. Compressed data cannot cross pixel row boundaries.
2. The number of uncompressed bytes per row is defined as the value in the ImageWidth tag, plus 7, divided by 8. If the resulting image map has an even number of bytes per row, the decompression buffer should be word-aligned.

### 4.2.5 TIFF Software

In this section we present two procedures for manipulating TIFF images. The procedure named SHOW_TIFF can be used to display a bitmap encoded in TIFF bilevel format. This procedure requires that the user pass a formatted data block, as shown in the header. This procedure calls the procedure named LOAD_TIFF, also listed in this section, which decompresses and loads the encoded image. The procedures are designed so as to place the TIFF file and the image bitmap in a separate data segment, therefore freeing the caller's code from having to devote storage space to TIFF data.

```
;*****************************************************************
;*****************************************************************
;                  TIFF File Access Procedures
;*****************************************************************
;*****************************************************************
;
;*****************************************************************
;                  segment for TIFF data
;*****************************************************************
TIFF_DATA       SEGMENT
;*********************|
; storage for TIFF file|
;*********************|
; Maximum size of TIFF file is 20K
TIFF_FILE       DB      20480 DUP (00H)
;*********************|
;   disk file buffer   |
;*********************|
DATA_BUF        DB      128 DUP (00H)   ; Disk data storage area
                DW      0
;*********************|
;    copy of caller's  |
;     display block    |
;*********************|
USER_BLOCK      DW      0       ; x coordinate
                DW      0       ; y coordinate
                DW      0       ; Offset of bitmap from start
                                ; of file
                DW      0       ; Number of pixel rows
                DB      0       ; Number of bytes per row
                DB      0       ; I R G B color code
                DB      15 DUP (00H)
                DW      0
;*********************|
;       bit image      |
;*********************|
BIT_IMAGE       DB      38400 DUP (00H)
;
TIFF_DATA       ENDS
;
;*****************************************************************
;                         code segment
;*****************************************************************
;
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE
;*****************************************************************
;*****************************************************************
;                        code segment data
;*****************************************************************
;*****************************************************************
```

```
; Code segment variables used by procedures
X_COORD         DW      0000H    ; Storage for x coordinate
BYTES           DB      0H       ; Number of bytes per block row
COUNT_8         DB      8        ; Bit counter for the
                                 ; VARI_PATTERN procedure
PIX_ROWS        DW      0        ; Number of pixel rows in map
;
; Data variables used by the SHOW_TIFF procedure
STRIP_OFFSET    DW      0        ; Offset of bitmap from start of
                                 ; TIFF file
TIFF_HANDLE     DW      0        ; File handle for TIFF disk file
IMAGE_SIZE      DW      0        ; Image dimension
EXP_COUNT       DW      0        ; Byte counter for expansion
USERS_DS        DW      0        ; Caller's DS segment
;
;****************************************************************
;               procedure to display raster image
;                         in TIFF format
;****************************************************************
;
SHOW_TIFF       PROC    NEAR
; Procedure to display a bit-mapped graphics file in TIFF format
; in VGA mode number 18
;
; On entry:
;       SI => caller's display block formatted as follows:
;
;   OFFSET    STORAGE    CONTENTS
;     0        WORD      x screen coordinate for image
;     2        WORD      y screen coordinate for image
;     4        WORD      Offset of bitmap from start of file
;                        (from the StripOffset tag)
;     6        WORD      number of vertical rows in bitmap
;                        (from the ImageLength tag)
;     8        BYTE      number of horizontal bytes in bitmap
;     9        BYTE      I R G B color code for bit display
;     10      STRING     ASCIIZ string containing the filename
;                        for the TIFF file
;
;
        CALL    LOAD_TIFF        ; Local procedure to load
                                 ; TIFF file into RAM and
                                 ; decompress
                                 ; image
        LEA     SI,USER_BLOCK    ; Reset entry pointer
; Initialize registers
        MOV     CX,WORD PTR [SI]        ; x coordinate
        MOV     CS:X_COORD,CX           ; Store in variable
        ADD     SI,2                    ; Bump pointer
        MOV     DX,WORD PTR [SI]        ; y coordinate
        ADD     SI,2                    ; Bump pointer
        MOV     BP,WORD PTR [SI]        ; Bitmap offset
        ADD     SI,2
        MOV     AX,WORD PTR [SI]        ; Number of pix rows
        MOV     CS:PIX_ROWS,AX          ; Store in variable
        ADD     SI,2                    ; Bump pointer
        MOV     BH,BYTE PTR [SI]        ; Bytes per block
        MOV     CS:BYTES,BH             ; Store in variable
        INC     SI                      ; Bump pointer
        MOV     AL,[SI]         ; Color code to AL
        MOV     CS:COUNT_8,8    ; Prime bit counter
        LEA     DI,BIT_IMAGE    ; Pointer to unpacked bitmap
; Register and variables after routine initialization:
;               CX = x coordinate of block start
;               DX = y coordinate of block start
;               BP = offset from start of TIFF file to bitmap
;               BH = number of bytes per block row
;               AL = color code
;               DI => image bitmap
;               CS:X_COORD = current x coordinate
;               CS:BYTES = number of bytes per block row
;               CS:PIX_ROWS = number of pixel rows in block
;               CS:COUNT_8 = 8
;**********************|
;   reset ES segment   |
;**********************|
        PUSH    AX              ; Save accumulator
        MOV     AX,0A000H       ; Video buffer segment base
        MOV     ES,AX           ; To extra segment
        POP     AX              ; Restore AX
;**********************|
;  display image block |
;**********************|
DISPLAY_BYTE_T:
        MOV     AH,[DI]         ; High nibble to AH
; Test for all zero display pattern
        CMP     AH,0            ; Nothing to display?
        JNE     TEST_BIT_T      ; Continue if not zero
        ADD     CX,8            ; Skip this byte
        JMP     NEXT_BYTE_T     ; Skip byte
TEST_BIT_T:
        TEST    AH,10000000B    ; Is high bit set?
        JZ      NEXT_BIT_T      ; Bit not set
; Set the pixel
        PUSH    AX              ; Save entry registers
        PUSH    BX
        CALL    PIXEL_ADD_18    ; Calculate pixel address in VGA
                                ; mode 18
        CALL    WRITE_PIX_18    ; Display pixel
        POP     BX              ; Restore registers
        POP     AX
NEXT_BIT_T:
        SAL     AH,1            ; Shift AH to test next bit
        INC     CX              ; Bump x coordinate counter
        DEC     CS:COUNT_8      ; Bit counter
        JZ      NEXT_BYTE_T     ; Exit if counter rewound
        JMP     TEST_BIT_T      ; Continue
; Index to next byte in row, if not at end of row
NEXT_BYTE_T:
        DEC     BH              ; Bytes per row counter
        JZ      NEXT_ROW_T      ; End of graphic row
BYTE_ENTRY_T:
        INC     DI              ; Bump graphic code pointer
        MOV     CS:COUNT_8,8    ; Reset bits counter
        JMP     DISPLAY_BYTE_T
; Index to next row
NEXT_ROW_T:
; Test for last graphic row
        DEC     CS:PIX_ROWS     ; Row counter
        JZ      GRAPH_END_T     ; Done, exit
        MOV     BH,CS:BYTES     ; Reset bytes counter
        INC     DX              ; Bump y coordinate control
        MOV     CX,CS:X_COORD   ; Reset x coordinate control
        JMP     BYTE_ENTRY_T
GRAPH_END_T:
        MOV     AX,CS:USERS_DS  ; Restore caller's DS
        MOV     DS,AX
        CLC                     ; No error reported
        RET
SHOW_TIFF       ENDP
;****************************************************************
;
LOAD_TIFF       PROC    NEAR
; procedure for loading a bilevel TIFF file
; On entry:
;         SI == caller's display block formatted as follows:
;
;   OFFSET    STORAGE    CONTENTS
;     0        WORD      x screen coordinate for image
;     2        WORD      y screen coordinate for image
;     4        WORD      Offset of bitmap from start of file
;                        (from the StripOffset tag)
;     6        WORD      number of vertical rows in bitmap
;     8        BYTE      number of horizontal bytes in bitmap
;     9        BYTE      I R G B color code for bit display
;     10      STRING     ASCIIZ string containing the filename
;                        for the TIFF file
;
; Assumptions and limitations:
;    1. Bitmap uncompressed or in PackBits compression mode
;    2. Unpacked bitmap not to exceed 255 pixel rows
;    3. TIFF bilevel image
;    4. Target display system resolution not to exceed 64K
```

```
;*********************|
;     ES to local data |
;*********************|
        PUSH    ES                 ; Save video buffer base
        MOV     AX,TIFF_DATA
        MOV     ES,AX
; Move user's display block to TIFF_DATA segment
        MOV     CX,25              ; 25 bytes in block
        LEA     DI,ES:USER_BLOCK
MOVE_TO_USER:
        MOV     AL,[SI]            ; Get caller's byte
        MOV     ES:[DI],AL         ; Move to local segment
        INC     SI                 ; Bump pointers
        INC     DI
        LOOP    MOVE_TO_USER
;*********************|
;   DS to local data   |
;*********************|
        MOV     AX,DS              ; Caller's DS to AX
        MOV     CS:USERS_DS,AX     ; Store in variable
; Change DS to local segment
        MOV     AX,TIFF_DATA       ; Local segment
        MOV     DS,AX
        ASSUME  DS:TIFF_DATA       ; New ASSUME statement
        POP     ES                 ; Restore video buffer base to ES
        LEA     SI,USER_BLOCK      ; Reset entry pointer
;*********************|
; store strip offset   |
;*********************|
        MOV     AX,WORD PTR [SI+4]         ; From display block
        MOV     CS:STRIP_OFFSET,AX         ; Store in CS variable
;*********************|
; calculate image size |
;*********************|
; Image dimensions are stored in the display block
; The image size is required during bitmap decompression
        MOV     AX,WORD PTR [SI+6]         ; Number of pixel rows
        MOV     BL,BYTE PTR [SI+8]         ; Number of bytes
        MOV     BH,0               ; Clear high of multiplier
        MUL     BX                 ; AX:DX = AX * BX
        MOV     CS:IMAGE_SIZE,AX           ; Store in CS variable
; Note that the high-order byte of the product can be discarded
; since code assumes that the display resolution is less than
; 64K
;********************|
;    open TIFF file   |
;********************|
        ADD     SI,10              ; Index to ASCIIZ string area
        MOV     DX,SI              ; For OPEN_FILE procedure
        CALL    OPEN_FILE          ; Procedure in SOLUTION.LIB
        JNC     OK_TIFF_OPEN       ; Continue if no carry
```

```
;**********************|
;  open operation fail |
;**********************|
        POP     SI                ; Restore context
        MOV     AX,CS:USERS_DS    ; Restore caller's DS
        MOV     DS,AX
        STC                       ; Code for error return
        RET
;**********************|
;    set ES segment    |
;    to local data     |
;**********************|
; Note: The SHOW_TIFF procedure uses the segment TIFF_DATA for
;       storing TIFF image data
        MOV     AX,TIFF_DATA      ; Set to local segment
        MOV     ES,AX
OK_TIFF_OPEN:
;**********************|
;   read TIFF file     |
;      into RAM        |
;**********************|
        LEA     DI,TIFF_FILE      ; Storage buffer for file
        MOV     CS:TIFF_HANDLE,AX         ; Store file handle
NEW_128:
        MOV     BX,CS:TIFF_HANDLE
        LEA     DX,DATA_BUF       ; Buffer for data storage
        PUSH    DI                ; Save buffer pointer
        CALL    READ_128          ; Read sector into buffer
        POP     DI                ; Restore buffer pointer
        CMP     AX,0              ; Test for end of file
        JNE     MOVE_128          ; Go if not at end of file
;**********************|
;    end of file       |
;**********************|
        MOV     BX,CS:TIFF_HANDLE         ; File handle
        CALL    CLOSE_FILE        ; Library routine
        JMP     END_OF_READ
;**********************|
;   move sector to     |
;    TIFF buffer       |
;**********************|
; At this point DATA_BUF holds 128 bytes from disk file
; DI — storage position in the TIFF file RAM buffer
MOVE_128:
        MOV     CX,128            ; Byte counter
        LEA     SI,DATA_BUF       ; Pointer to data just read
PLACE_128:
        MOV     AL,[SI]           ; Byte from DATA_BUF
        MOV     [DI],AL           ; Into font's buffer
        INC     SI                ; Bump pointers
        INC     DI
        LOOP    PLACE_128       ; Continue until all sector read
; At this point the 128 bytes newly read from the disk file are
; stored in the font's buffer
        JMP     NEW_128
;****************************|
;        unpack bitmap       |
;****************************|
; Unpacking logic for TIFF PackBits scheme
; PackBits packages consist of 2 bytes. The first byte (n)
; encodes the following options:
;       1. if n is in the range 0 to 127, then next n+1 bytes are
;          to be interpreted as literal values
;       2. if n is in the range -127 to -1, then the following
;          byte is to be repeated -n+1 times
;       3. if n = 128, then no operation is executed
END_OF_READ:
        MOV     CS:EXP_COUNT,0  ; Clear counter
        LEA     DI,BIT_IMAGE    ; Destination in display block
        LEA     SI,TIFF_FILE    ; Pointer to start of file
        ADD     SI,CS:STRIP_OFFSET ; Add offset to bitmap
; SI == start of image if there is a single strip
TEST_N_BYTE:
        MOV     AL,[SI]         ; Get n byte
        CMP     AL,128          ; Code for NOP
        JB      LITERAL_CODE    ; Go if in the literal range
        JA      REPEAT_CODE     ; Go if in repeat range
; Code is 128 (NOP)
        INC     SI              ; Skip NOP code
        JMP     NEXT_PACK_CODE  ; Continue
;*********************|
;   literal expansion |
;*********************|
LITERAL_CODE:
        MOV     CL,AL           ; Counter to CL
        MOV     CH,0            ; Clear high byte of counter
        INC     CX              ; Add 1
        INC     SI              ; Skip n byte
        ADD     CS:EXP_COUNT,CX ; Add bytes to counter
LIT_MOVE:
        MOV     AL,[SI]         ; Get literal byte
        NOT     AL              ; Invert white and black bits
        MOV     [DI],AL         ; Place in bitmap
        INC     DI              ; Bump pointers
        INC     SI
        LOOP    LIT_MOVE
        JMP     NEXT_PACK_CODE
;*********************|
;  repeated expansion |
;*********************|
REPEAT_CODE:
        NEG     AL              ; Negate to convert 2's
```

```
                                ; complement representation
        MOV     CL,AL           ; Counter to CL
        MOV     CH,0            ; Clear high byte of counter
        INC     CX              ; Add 1
        INC     SI              ; Skip n byte
        ADD     CS:EXP_COUNT,CX ; Add bytes to counter
        MOV     AL,[SI]         ; Get byte to repeat
        NOT     AL              ; Invert black and white bits
        INC     SI              ; Skip to next n byte
EXP_MOVE:
        MOV     [DI],AL         ; Place byte in buffer
        INC     DI              ; Bump bitmap pointer
        LOOP    EXP_MOVE
;**********************|
;  get next pack code  |
;**********************|
; CS:EXP_COUNT holds the total bytes in bitmap at this point
; CS:IMAGE_SIZE holds the total bytes in the expanded bitmap
NEXT_PACK_CODE:
        MOV     AX,CS:EXP_COUNT    ; Bytes now in bitmap
        CMP     AX,CS:IMAGE_SIZE   ; Compare with map size
        JAE     DISPLAY_IMAGE      ; Go if at end of image
        JMP     TEST_N_BYTE
DISPLAY_IMAGE:
        RET
LOAD_TIFF       ENDP
;****************************************************************
;                        auxiliary procedures
;****************************************************************
;
CLOSE_FILE      PROC    NEAR
; Close file using file handle
; On entry:
;        BX = file handle
; On exit:
;        carry clear if operation successful - file closed
;        carry set if operation failed - invalid handle or file
;        not open
;
        MOV     AH,62           ; DOS service request
        INT     21H
        RET
CLOSE_FILE      ENDP
;****************************************************************
;
READ_128        PROC    NEAR
; Read 128 bytes from an open file into buffer using the file
; handle. This procedure assumes that the file has been
; previously opened or created using the procedure OPEN_CREATE
;
; On entry:
```

```
;         BX =   file handle
;         DX — 128 bytes user buffer
; On exit:
;         carry clear if operation successful
;           AX = number of bytes read into buffer
;           AX = 0 if end of file
;         carry set if operation failed
;           AX = error code
;                5 = access denied
;                6 = invalid handle or file not open
;
        PUSH    CX              ; Save entry CX
        MOV     AH,63           ; DOS service request
        MOV     CX,128          ; No. of bytes to read
        INT     21H
        POP     CX              ; Restore
        RET
READ_128        ENDP
;****************************************************************
;
OPEN_FILE       PROC    NEAR
; Open file using an ASCIIZ string for the filename
; On entry:
;         DX — buffer containing ASCIIZ string for filename
; On exit:
;         if carry clear file was opened successfully
;         AX = file handle
;         if carry set open operation failed
;         AX = error code
;              1 = invalid function
;              2 = file not found
;              3 = path not found
;              4 = no available handle
;              5 = access denied
;             12 = invalid access code
;
        MOV     AH,61           ; DOS service request number
                                ; to open file (handle mode)
        MOV     AL,2            ; Read/write access
        INT     21H
        RET
OPEN_FILE       ENDP
;
;****************************************************************
;               VGA device drivers for mode number 18
;****************************************************************
;
PIXEL_ADD_18    PROC     NEAR
; Address computation from x and y pixel coordinates
; On entry:
;                CX = x coordinate of pixel (range 0 to 639)
;               DX = y coordinate of pixel (range 0 to 479)
; On exit:
;               BX = byte offset into video buffer
;               AH = bit mask for the write operation using
;                    VGA write modes 0 or 2
;               AL is preserved
; Save all entry registers
        PUSH    CX
        PUSH    DX
; Compute address
        PUSH    AX              ; Save accumulator
        PUSH    CX              ; Save x coordinate
        MOV     AX,DX           ; y coordinate to AX
        MOV     CX,80           ; Multiplier (80 bytes per row)
        MUL     CX              ; AX = y times 80
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
; Prepare for division
        MOV     CL,8            ; Divisor
        DIV     CL              ; AX / CL = quotient in AL and
                                ; remainder in AH
; Add in quotient
        MOV     CL,AH           ; Save remainder in CL
        MOV     AH,0            ; Clear high byte
        ADD     BX,AX           ; Offset into buffer to BX
        POP     AX              ; Restore AX
; Compute bit mask from remainder
        MOV     AH,10000000B ; Unit mask for 0 remainder
        SHR     AH,CL           ; Shift right CL times
; Restore all entry registers
        POP     DX
        POP     CX
        RET
PIXEL_ADD_18    ENDP
;****************************************************************
;
WRITE_PIX_18    PROC    NEAR
; VGA mode number 18 device driver for writing an individual
; pixel or a pixel pattern to the graphics screen
;
; On entry:
;            ES:BX = byte offset into the video buffer
;               AL = pixel color in IRGB format
;               AH = bit pattern to set
;
; This routine assumes that write mode 2 has been set
;
; Note: programs using this procedure usually precede their call
;       by one to ES_TO_APA (to set the video segment base) and
;       another one to PIXEL_ADD_18 (to obtain the byte offset
;       and pixel mask).
;       This procedure does not reset the default write mode nor
;       the contents of the Bit Mask register
;
        PUSH    DX          ; Save outer loop counter
        PUSH    AX          ; Color byte
        PUSH    AX          ; Twice
; Set Bit Mask Register according to mask in AH
        MOV     DX,3CEH     ; Graphic controller latch
        MOV     AL,8
        OUT     DX,AL       ; Select data register 8
        JMP     SHORT $+2
        INC     DX          ; To 3CFH
        POP     AX          ; AX once from stack
        MOV     AL,AH       ; Bit pattern
        OUT     DX,AL       ; Load bit mask
        JMP     SHORT $+2
; Write color code
        MOV     AL,ES:[BX]  ; Dummy read to load latch
                            ; registers
        POP     AX          ; Restore color code
        MOV     ES:[BX],AL  ; Write the pixel with the
                            ; color code in AL
        POP     DX          ; Restore outer loop counter
        RET
WRITE_PIX_18    ENDP

CODE    ENDS
        END
```

## 4.3 Bitmap Image Acquisition

Many applications require bit-mapped images to serve as graphics objects or as background. The program designer can generate or acquire these images through the following means:

1. By creating an image bitmap in the application's memory space. For example, the bitmap in Figure 4.2 can be defined as an array of 60 bytes, 30 words, or 15 doublewords in RAM.
2. By using a paint or draw program to create a graphics image and then saving this image in a standard image file format that can be manipulated by the application's code.
3. By scanning an existing image and saving the resulting scan in a standard image file format.
4. By using an existing image file. If the image file is proprietary, its use may require the purchase of reproduction or other rights from the copyright holder. However, if the image is in the public domain, no reproduction right, are necessary for its use. Shareware images are in a special category which is subject to special rules.

5. By digitally and mathematically operating on an existing image or image description. For example, a ray-tracing program can create or enhance an existing image by applying principles of optics.
6. By a combination of two or more of the above methods, for example, scanning an image and then modifying the scanned bitmap by means of a bitmap editing or ray-tracing program.

Figure 4.16 shows the evolution of a digitized image from the original scanned bitmap, which is retouched to eliminate the space shuttle's external tank and rocket boosters, and then vectorized into a geometrical image which can be scaled and rotated without distortion.

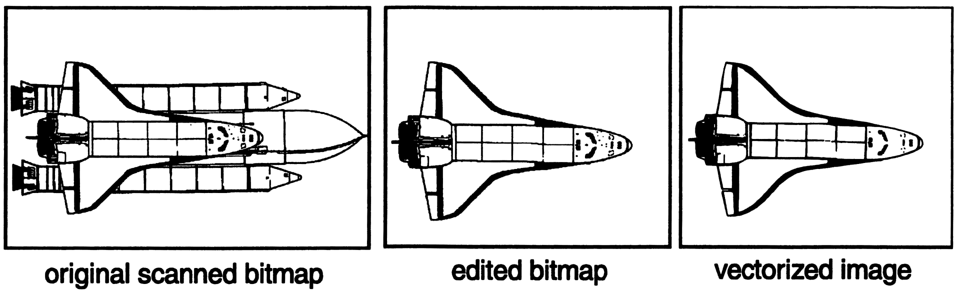


Figure 4.16 *Image Acquisition Process*

### 4.3.1 Legal Considerations

The use of existing images can give rise to legal questions. In principle, the creator of the image has the right to reproduce it and to prevent others from doing so. The right to reproduce or make copies (copyright) falls in the category of intellectual property rights. The law requires that for these rights to be effective the creator must post a copyright notice. This notice must meet certain legal requirements; namely, it must contain the word "copyright," the copyright symbol, the year of creation, and the word "by" followed by the author's name. For example:

Copyright © 1994 by J. Smith.

The details of copyright protection are contained in the *Copyright Act of 1976*. In many cases regarding conventional works of literature or graphics arts the provisions of the copyright law are relatively certain and well defined. Not so in the digital environment, which has originated many new problems of interpretation and extension of copyright, not all of which are definitively solved. One field in which there is still considerable doubt is that of derivative works. A derivative work occurs when a creation is based or inspired by another one. Since the copyright law requires, but does not define, "originality," the question is when and if a derivative work becomes another original. For example, if a photograph is scanned and modified by software, is the derived result an original work? If so, how much editing is necessary to make the derivative work a new original?

Another field in which many legal questions still remain is in the compilation of data. Since copyright requires that the work be an original creation, it is valid to question if raw data meets the originality requirement. For example, is the data contained in a telephone directory protected by copyright? Until 1991 the courts had held a sweat-of-the-brow theory of originality that included works of compilation, such as a telephone directory. However, in the case *Feist Publications vs. Rural Telephone Service* the United States Supreme Court rejected this theory and required that some degree of "originality in selection and arrangement" be present in the work in order to make a work eligible for copyright protection.

In the field of computer graphics imagery the problems of copyright protection still pose many unresolved questions. To the developer the only available guidelines are the elements of originality, creativity, and authorship required by the Copyright Act of 1976, and also the fact that in copyright infringement lawsuits the plaintiff is required to prove that the defendant copied expressive elements in the original work. The element of "substantial similarity" is also often considered in this context. Therefore, we can conclude that if expressive elements of the original image are copied to a degree that makes the derived image substantially similar to the original one, then the derived image is not an original work and cannot claim copyright.

Part

2

# Animation Programming

Chapter

5

# Animation in VGA Graphics

## 5.0 The VGA Standard

*Video Graphics Array* (VGA) was introduced in 1987 with the IBM PS/2 line. *Multi-Color Graphics Array* (MCGA), an under-featured version of VGA, was furnished with the lower-end PS/2 machines Models 25 and 30. Since then VGA has been the standard PC video system.

The main technological innovation introduced by VGA was a change from digital to analog video display driver technology. The reason is that analog monitors can produce a much larger color selection than digital ones. VGA graphics hardware includes a digital-to-analog converter, usually called the DAC, and 256K of video memory. The DAC outputs the red, green, and blue signals to the analog display. Video memory is divided into four 64K video maps, called the *bit planes*. VGA supports all the display modes available in its predecessors, MDA, CGA, and EGA. In addition, it creates several new alphanumeric and graphics modes. The most interesting of the new standard graphics modes are mode 18, with 640-by-480 pixel resolution in 16 colors, and mode 19, with 320-by-200 pixel resolution in 256 colors. The effective resolution of the VGA text modes is 720-by-400 pixels. These text modes can execute in 16 colors or in monochrome. Three different fonts can be selected in the alphanumeric modes.

In the VGA access to the video system registers and to video memory is through the system microprocessor. The microprocessor read and write operations to the video buffer are automatically synchronized by the VGA hardware with the CRT controller so as to eliminate interference. For this reason VGA programs, unlike those for the CGA, can access video memory at any time without fear of introducing screen snow or other unsightly effects.

### 5.0.1 VGA Characteristics

The resolution of a video graphics system is measured in the total number of separately addressable elements per unit area, called screen *pixels*. Resolution is measured in pixels per inch. The maximum resolution of a VGA system is approximately 80 pixels per inch, both vertically and horizontally. In VGA this density is determined by a screen structure of 640 pixels per each 8-inch screen row and 480 vertical pixels per each 6-inch screen column. But not all video systems output a symmetrical pixel density. For example, the maximum resolution of the EGA standard is the same as that of the VGA on the horizontal axis (80 pixels per inch) but only 58 pixels per inch on the vertical axis.

The asymmetrical pixel grid of the EGA and of other less refined video standards introduced programming complications. For example, in a symmetrical VGA screen a square figure can be drawn using lines of the same pixel length, but these lines would produce a rectangle in an asymmetrical system. By the same token, the pixel pattern of a circle in a symmetrical system appears as an ellipse in an asymmetrical one.

The major limitations of the VGA system are resolution, color range, and performance. VGA density of 80 pixels per inch is a substantial improvement in relation to its predecessors, the CGA and the EGA, but still not very high when compared to the 600 dots per inch of a state-of-the-art laser printer or the 1200 and 2400 dots per inch of a quality color plate. The low resolution is one reason why VGA screen images are often not lifelike; bitmaps appear grainy and we can often detect that geometrical figures consist of straight line segments. VGA can display up to 256 simultaneous colors. However, the 256 colors are not available in the modes with the best resolution. Therefore, the VGA programmer must chose between 80 pixels per inch resolution in 16 colors (mode 18) or 40 pixels per inch resolution in 256 colors (mode 19).

But, to the animation programmer, perhaps the greatest limitation of the VGA standard is its performance. The video display update operations in VGA detract from general system efficiency, since it is the microprocessor that must execute all video read and write operations. In the second place, the video functions execute slowly when compared to dedicated graphics work stations. This is particularly noticeable in the graphics modes, in which a full screen redraw can take several seconds. For this reason, animated programs that are not carefully designed execute in VGA system with a jolting effect that is unnatural and visually disturbing.

### 5.0.2 VGA Standard Modes

The original video systems used in the PC, such as CGA, MDA, and EGA, had monitor-specific modes. For example, the CGA turns the color burst off in modes 0, 2, and 4 and on in modes 1, 3, and 5. Mode number 7 is available in the MDA and in the EGA equipped with a monochrome display, but not in the CGA or EGA systems equipped with color monitors. In the VGA standard, on the other hand, the video modes are not dependent on the monitor. For example, a VGA equipped with a direct drive color monitor can execute in monochrome mode 7. Table 5.1 lists the properties of the VGA video modes.

Table 5.1 *VGA Video Modes*

| MODE | TYPE | COLORS | PALETTE SIZE | BUFFER ADDRESS | | CHAR. BOX | MAX. PAGES | VERT. FREQ. | RESOLUTION IN PIXELS |
|---|---|---|---|---|---|---|---|---|---|
| 0,1 | text | 16 | 256K | 40 x 25 | B8000H | 9 x 16 | 8 | 70 Hz | 360 x 400 |
| 2,3 | text | 16 | 256K | 80 x 25 | B8000H | 9 x 16 | 8 | 70 Hz | 720 x 400 |
| 4,5 | graphics | 4 | 256K | 40 x 25 | B8000H | 8 x 8 | 1 | 70 Hz | 320 x 200 |
| 6 | graphics | 2 | 256K | 80 x 25 | B8000H | 8 x 8 | 1 | 70 Hz | 640 x 200 |
| 7 | text | | | 80 x 25 | B0000H | 9 x 16 | 8 | 70 Hz | 720 x 400 |
| 13 | graphics | 16 | 256K | 40 x 25 | A0000H | 8 x 8 | 8 | 70 Hz | 320 x 200 |
| 14 | graphics | 16 | 256K | 80 x 25 | A0000H | 8 x 8 | 4 | 70 Hz | 640 x 200 |
| 15 | graphics | | | 80 x 25 | A0000H | 8 x 14 | 2 | 70 Hz | 640 x 350 |
| 16 | graphics | 16 | 256K | 80 x 25 | A0000H | 8 x 14 | 2 | 70 Hz | 640 x 350 |
| 17 | graphics | 2 | 256K | 80 x 30 | A0000H | 8 x 16 | 1 | 60 Hz | 640 x 480 |
| 18 | graphics | 16 | 256K | 80 x 30 | A0000H | 8 x 16 | 1 | 60 Hz | 640 x 480 |
| 19 | graphics | 256 | 256K | 40 x 25 | A0000H | 8 x 8 | 1 | 70 Hz | 320 x 200 |

The VGA buffer can start in any one of three possible addresses: B0000H, B8000H, and A0000H (see Table 5.1). Address B000H is used only when mode 7 is enabled; in this case VGA is emulating the Monochrome Display Adapter. In enhanced mode 7 the VGA displays its highest horizontal resolution (720 pixels) and uses a 9-by-16 dots text font. However, in this mode the VGA has no graphics capabilities. Buffer address A000H is active while VGA is in a graphics mode. Also note that video modes 17 and 18, with 480 pixel rows, were introduced with the VGA and MCGA standards. Therefore they are not available in CGA and EGA systems. These modes produce a symmetrical pixel density of 640-by-480 screen dots. Mode 19 has 256 simultaneous colors, the most extensive one in the VGA standard, however, its horizontal resolution is half of the one in mode number 18. In addition, VGA mode 19 has a resolution of 320-by-200 pixels. This creates a nonsymmetrical pixel grid; in other words, the screen aspect ratio is not 1:1, as it is in mode number 18. In this environment the program code must generate a rectangle in order to display a square and an ellipse in order to display a circle.

### 5.0.3 VGA Nonstandard Modes

Graphics and animation programmers have tinkered with the VGA in an effort to create display modes that better suit their own purposes and requirements. The best known nonstandard VGA mode is the one called Mode X. Although this mode has been in use by graphics and animations programmers for some time, it was fist documented in an article by Michael Abrash published in *Dr. Dobb's Journal* in July 1991 (see Bibliography). This mode is not documented or supported in IBM's technical documents for VGA or by other major VGA manufacturers.

VGA mode X has a resolution of 320-by-240 pixels in 256 colors. It displays 40 more pixel rows than VGA standard mode 19 (see Table 5.1). But these additional 40 pixel rows are not achieved easily. In the first place the screen space in mode number 19 consists of 64,000 pixels ($320 \times 200$). This number of pixels can be contained in a single segment or video map. Expanding the display to 240 rows raises the total number of screen pixels to 76,800 (320 × 240), which exceeds the capacity of a processor segment register and, therefore, of a single video map. On the other hand, the 320-by-240 resolution provides a symmetrical drawing grid, with the advantages mentioned in Section 5.0.2.

Several features of VGA mode X make it attractive to the animations programmer. In the first place it offers a better resolution than the only other mode in 256 colors (mode 19). Mode X operates on a symmetrical pixel grid, which simplifies programming. Also, mode X allows page flipping, which is not the case with modes 18 and 19. Finally, the performance of mode X considerably exceeds that of VGA mode 19. For these reasons VGA mode X is considered in detail in this and other chapters.

## 5.1 VGA Architecture

The VGA system is divided into three separate components: the VGA chip, video memory, and a digital-to-analog converter (DAC). Figure 5.1 shows the interconnections between the elements of the VGA system.

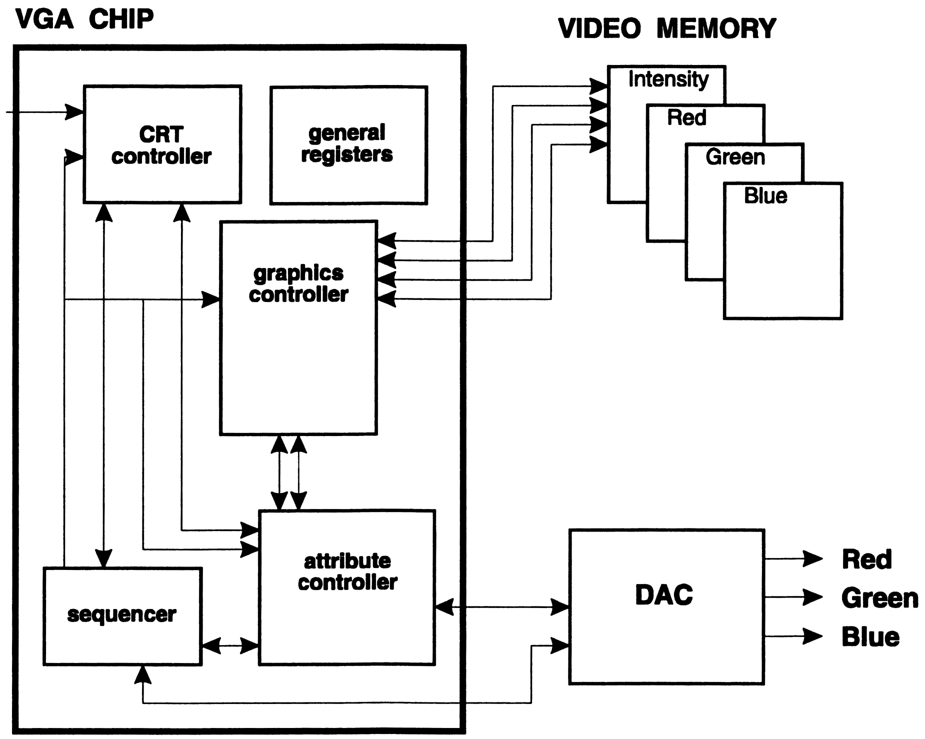


Figure 5.1 *VGA Component Diagram*

### 5.1.1 Video Memory

All VGA systems contain the 256K of video memory that is part of the hardware. This memory is logically arranged in four 64K blocks. In some modes these blocks form the video maps (labeled blue, green, red, and intensity in Figure 5.1). The four maps are sometimes referred to as bit planes 0 to 3.

#### Alphanumeric Modes

In the alphanumeric modes 0, 1, 2, 3, and 7 (see Table 5.1) the VGA video buffer is structured to hold character codes and attribute bytes. The VGA standard allows redefining two of the attribute bits in the color alphanumeric modes: bit 7 can be redefined to control the background intensity, and bit 3 can be redefined to perform a character-set select operation. Figure 5.2 shows the VGA attribute byte in the monochrome and color alphanumeric modes.

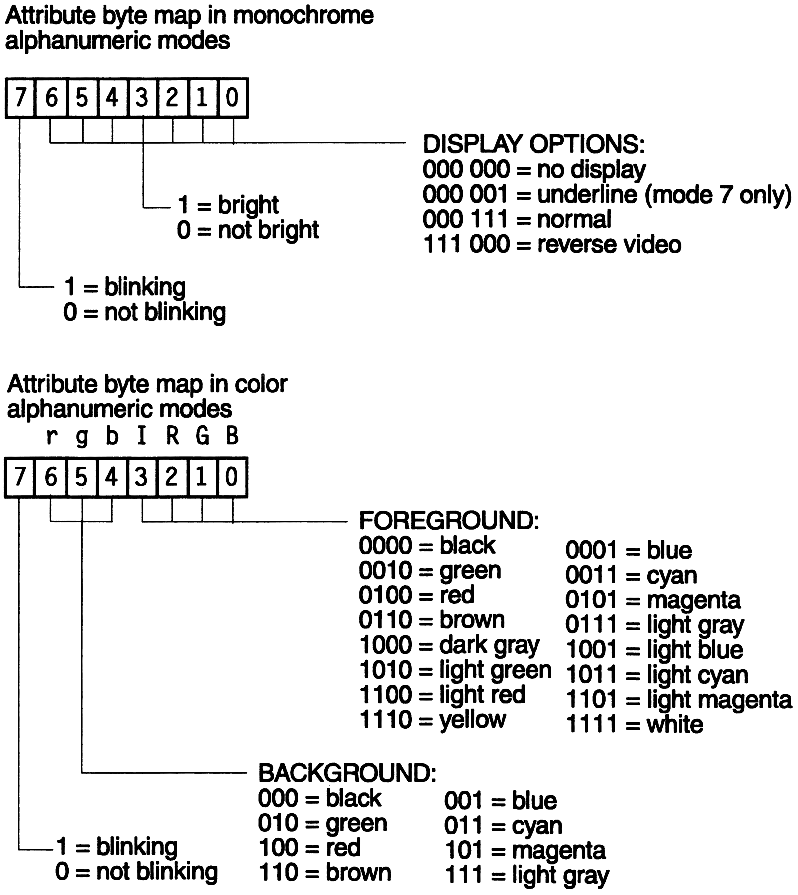


Figure 5.2 *VGA Attribute Byte Maps*

Bit 3 of the attribute byte controls the foreground intensity in both monochrome and color systems. Alternatively this bit can be used to select one of the character sets provided in the BIOS. The default function of bit 7 is the blink function. However, bit 3 can be reprogrammed to control the foreground intensity. The programmer can toggle the functions assigned to bits 3 and 7 of the attribute byte by means of BIOS service calls or by programming the VGA registers directly.

## Graphics Modes

One of the problems confronted by the designers of the VGA system was the limited memory space of IBM microcomputers under MS-DOS. Recall that in VGA mode number 18 the video screen is composed of 480 rows of 640 pixels per row, for a total of 307,200 screen pixels. If eight pixels are encoded per memory byte, each color map would take up approximately 38K. This means that the four maps required to encode 16 colors would need approximately 154K. The VGA designers were able to reduce this memory space by using a latching mechanism that maps all four color maps to the same memory area. Figure 5.3 shows the latching mechanism in VGA mode 18.

In Figure 5.3 we can see how the color of a single screen pixel is stored in four memory maps. Logically, the four maps are located at the same address. In mode 18 the base address for the video maps is A0000H. Which map is active depends on which of the four latches is open. Notice that the color codes for the first eight screen pixels are stored in the four maps labeled Intensity, Red, Green, and Blue. The first screen pixel has the intensity bit and the green bit set; therefore it appears light green.

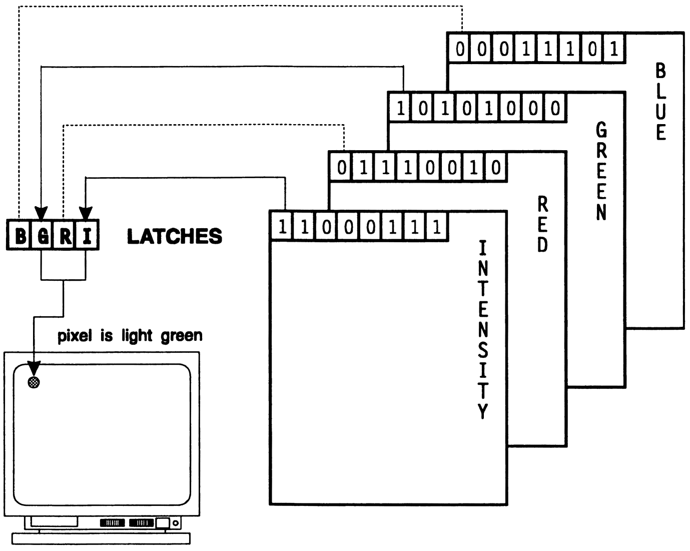


Figure 5.3 *Memory Structure in VGA Mode 18*

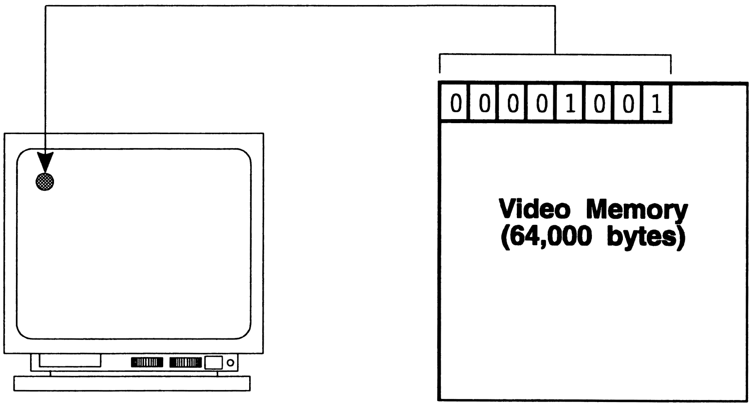


Figure 5.4 *Memory Structure in VGA Mode 19*

VGA memory mapping is different in the various alphanumeric and graphics modes. In Figure 5.3 we see that in mode number 18 the color of each screen pixel is determined by the bit settings in four memory maps. However, in mode number 19, in which VGA can display 256 colors, each screen pixel is determined by one video buffer byte. Figure 5.4 shows the memory mapping in VGA mode 19.

Although, to the programmer, the buffer appears as a linear space starting at address A000H, in reality VGA uses all four bit planes to store video data in mode 19. The color value assigned to each pixel in the 256-color modes depends on the DAC register setting, which is explained later in this chapter.

VGA mode X shares some of the characteristics of modes 18 and 19. Figure 5.5 shows the memory mapping in VGA mode X.

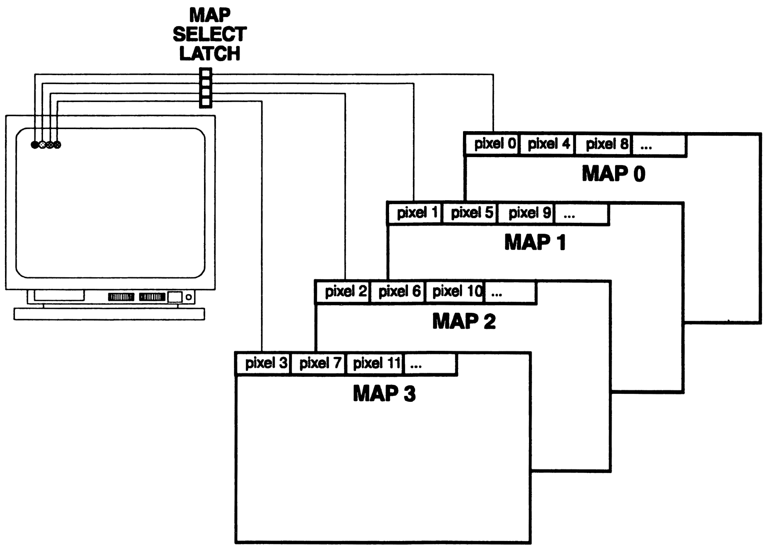

MAP
SELECT
LATCH
pixel 0 pixel 4 pixel 8 ...
MAP 0
pixel 1 pixel 5 pixel 9 ...
MAP 1
pixel 2 pixel 6 pixel 10 ...
MAP 2
pixel 3 pixel 7 pixel 11 ...
MAP 3

Figure 5.5 *Memory Structure in VGA Mode X*

Like mode 18, mode X is a planar mode; that is, video data is stored in several planes or maps. In mode X the four planes, which are located at the same physical address, are mapped to a different range of screen pixels. In Figure 5.5 we can see that the video data in map 0 (plane 0) is mapped to pixel number 0, and all successive pixels in an arithmetic sequence with a common difference of 4. Map 1 contains video data for pixel number 1, and all successive pixels in a sequence with a common difference of 4. The same applies to maps 2 and 3. Which pixel map is active depends on the latching mechanism, controlled by the VGA Map Mask register described later in this chapter. Mode X resembles mode 19 in that the color of a screen pixel is determined by a memory byte. This simplifies and speeds up processing since the time-consuming bit-masking operations necessary in mode 19 are not required in mode X.

Other VGA graphics modes in Table 5.1 were created merely to ensure compatibility with previous video systems. Specifically, VGA graphics modes 4, 5, and 6 are compatible with modes in the CGA, EGA, and PCjr; modes 13, 14, 15, and 16 are compatible with EGA; and graphics mode 17 (a two-color version of mode 18) was created for compatibility with the MCGA standard. The two proprietary VGA modes are mode 18 (640-by-480 pixels in 16 colors) and mode 19 (320-by-200 pixels in 256 colors). Mode X, as already mentioned is not a standard VGA mode. VGA graphics and animation deal mostly in these three modes. Therefore it is these modes that are described in detail.

## 5.2 The VGA Registers

In Figure 5.1 we see that the VGA system includes a chip containing several registers, a memory space dedicated to video functions, and a digital-to-analog converter, or DAC. The VGA registers are mapped to the system's address space and accessed by means of the central processor. The VGA programmable registers (excluding the DAC) belong to five groups shown in Table 5.2, namely:

1. The General registers. This group is sometimes called the *external registers* due to the fact that, on the EGA, they were located outside the VLSI chip. The General registers provide miscellaneous and control functions.
2. The CRT Controller registers. This group of registers controls the timing and synchronization of the video signal and also the cursor size and position.
3. The Sequencer registers. This group of registers controls data flow into the Attribute Controller, generates the timing pulses for the dynamic RAMs, and arbitrates memory accesses between the CPU and the video system. The Map Mask registers in the Sequencer allow the protection of entire memory maps.
4. The Graphics Controller registers. This group of registers provides an interface between the system microprocessor, the Attribute Controller, and video memory, while VGA is in a graphics mode.
5. The Attribute Controller registers. This group of registers determines the characteristics of the character display in the alphanumeric modes and the pixel color in the graphics modes.

Table 5.2 *VGA Register Groups*

| | | EMULATING | | |
|---|---|---|---|---|
| REGISTER | READ/ WRITE | MDA | CGA | EITHER |
| **GENERAL REGISTERS** | | | | |
| 1. Miscellaneous output | Write<br>Read | | | 03C2H<br>03CCH |
| 2. Input status 0 | Read | | | 03C2H |
| 3. Input status 1 | Read | 03BAH | 03DAH | |
| 4. Feature control | Write<br>Read | 03BAH | 03DAH | <br>03CAH |
| 5. Video subsystem enable | R/W | | | 03C3H |
| 6. DAC state | Read | | | 03C7H |
| **ATTRIBUTE CONTROLLER REGISTERS** | | | | |
| 1. Address | R/W | | | 03C0H |
| 2. Other | Write<br>Read | | | 03C0H<br>03C1H |
| **CRT CONTROLLER REGISTERS** | | | | |
| 1. Index | R/W | 03B4H | 03D4H | |
| 2. Other CRT controller | R/W | 03B5H | 03D5H | |
| **SEQUENCER REGISTERS** | | | | |
| 1. Address | R/W | | | 03C4H |
| 2. Other | R/W | | | 03C5H |
| **GRAPHICS CONTROLLER REGISTERS** | | | | |
| 1. Address | R/W | | | 03CEH |
| 2. Other | R/W | | | 03CFH |

### 5.2.1 VGA General Registers

The General registers are used primarily in initialization of the video system and in mode setting. Most applications let the system software handle the initialization of the video functions controlled by the General registers. For example, the easiest and most reliable way for setting a video mode is BIOS service number 0, of interrupt 10H. On the other hand, the code has to access the General registers when setting a nonstandard VGA mode, such as mode X previously mentioned. Figure 5.6 shows some programmable elements in the VGA General register group.

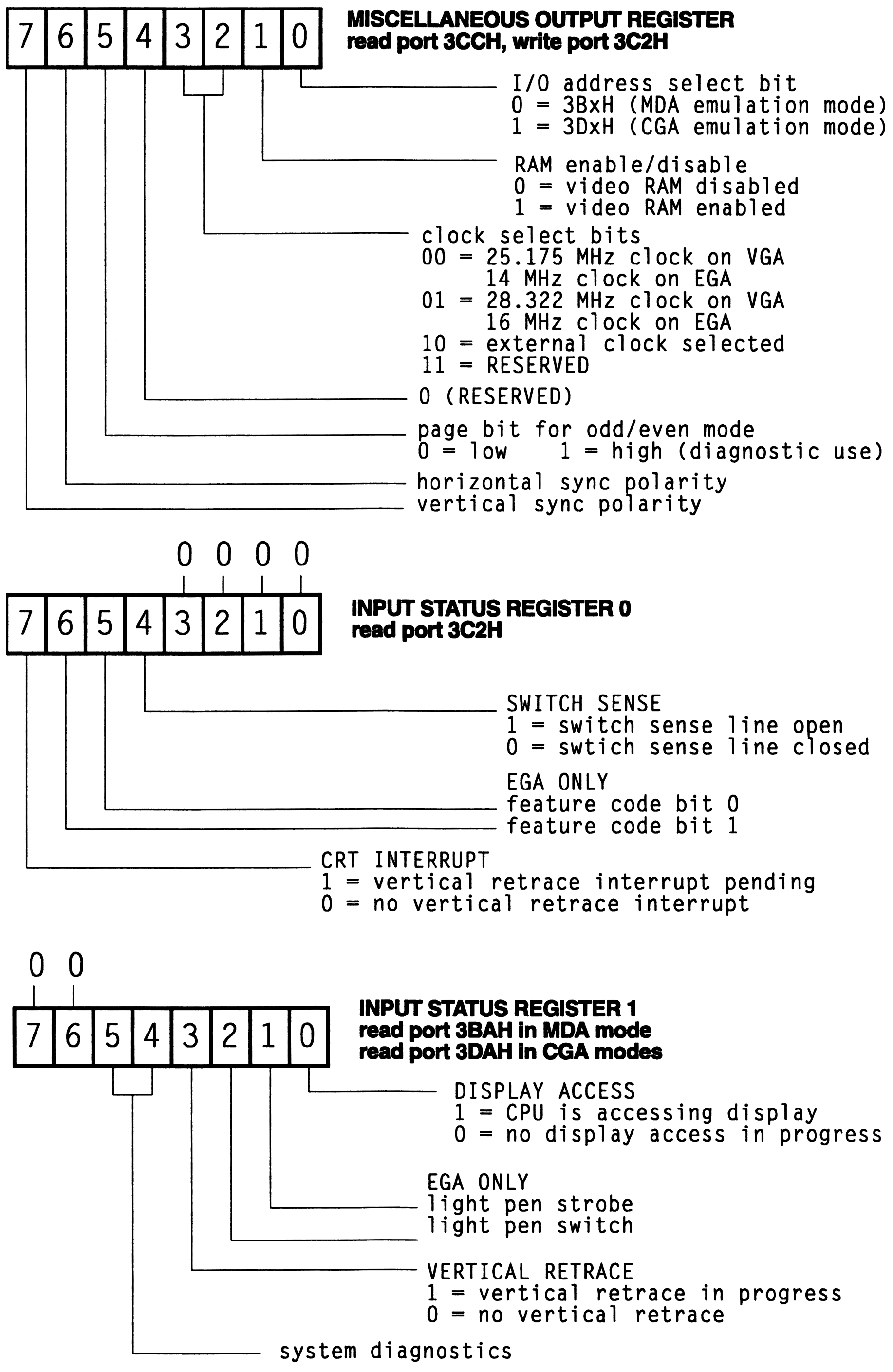

7 6 5 4 3 2 1 0
MISCELLANEOUS OUTPUT REGISTER
read port 3CCH, write port 3C2H
I/O address select bit
0 = 3BxH (MDA emulation mode)
1 = 3DxH (CGA emulation mode)
RAM enable/disable
0 = video RAM disabled
1 = video RAM enabled
clock select bits
00 = 25.175 MHz clock on VGA
14 MHz clock on EGA
01 = 28.322 MHz clock on VGA
16 MHz clock on EGA
10 = external clock selected
11 = RESERVED
0 (RESERVED)
page bit for odd/even mode
0 = low 1 = high (diagnostic use)
horizontal sync polarity
vertical sync polarity
0 0 0 0
7 6 5 4 3 2 1 0
INPUT STATUS REGISTER 0
read port 3C2H
SWITCH SENSE
1 = switch sense line open
0 = swtich sense line closed
EGA ONLY
feature code bit 0
feature code bit 1
CRT INTERRUPT
1 = vertical retrace interrupt pending
0 = no vertical retrace interrupt
0 0
7 6 5 4 3 2 1 0
INPUT STATUS REGISTER 1
read port 3BAH in MDA mode
read port 3DAH in CGA modes
DISPLAY ACCESS
1 = CPU is accessing display
0 = no display access in progress
EGA ONLY
light pen strobe
light pen switch
VERTICAL RETRACE
1 = vertical retrace in progress
0 = no vertical retrace
system diagnostics

Figure 5.6 *VGA General Registers*

Note that bit number 7 of Input Status Register 0, at port 3C2H, (see Figure 5.6) is used in determining the start of the vertical retrace cycle of the CRT controller. This operation is sometimes necessary to avoid interference when updating the video buffer. Animation routines also use the vertical retrace cycle to intercept a pulse that can serve to produce time-lapsed screen updates. The procedure named TIME_VR, listed in Chapter 7, performs this timing operation.

The Feature Control Register and the Video Subsystem Enable Register in the General Register group are reserved. IBM recommends that applications use subservice 12H of BIOS interrupt 10H to disable address decoding by the video subsystem. The DAC State Registers are discussed in Section 5.3.

### 5.2.2 VGA CRT Controller

The VGA CRT Controller register group is the equivalent of the Motorola 6845 CRT Controller chip of the PC line. When VGA is emulating the MDA, the port address of the CRT Controller is 3B4H; when it is emulating the CGA, then the port address is 3D4H. These ports are the same as those used by the MDA and the CGA cards. Table 5.3 lists the registers in the CRT Controller group.

Table 5.3 *VGA CRT Controller Registers*

| PORT | OFFSET | DESCRIPTION |
|---|---|---|
| 03x4H | | Address register |
| 03x5H | 0 | Total horizontal characters minus 2 (EGA)<br>Total horizontal characters minus 5 (VGA) |
| | 1 | Horizontal display end characters minus 1 |
| | 2 | Start horizontal blanking |
| | 3 | End horizontal blanking |
| | 4 | Start horizontal retrace pulse |
| | 5 | End horizontal retrace pulse |
| | 6 | Total vertical scan lines |
| | 7 | CRTC overflow |
| | 8 | Preset row scan |
| | 9 | Maximum scan line |
| | 10 | Scan line for cursor start |
| | 11 | Scan line for cursor end |
| | 12 | Video buffer start address, high byte |
| | 13 | Video buffer start address, low byte |
| | 14 | Cursor location, high byte |
| | 15 | Cursor location, low byte |
| | 16 | Vertical retrace start |
| | 17 | Vertical retrace end |
| | 18 | Last scan line of vertical display |
| | 19 | Additional word offset to next logical line |
| | 20 | Scan line for underline character |
| | 21 | Scan line to start vertical blanking |
| | 22 | Scan line to end vertical blanking |
| | 23 | CRTC mode control |
| | 24 | Line compare register |

**Notes:** 3x4H/3x5H = 3B4H/3B5H when emulating the MDA
3x4H/3x5H = 3D4H/3D5H when emulating the CGA

Most registers in the CRT Controller are modified only during mode changes. Since this operation is frequently performed by means of a BIOS service, most programs do not access the CRT Controller registers directly. One exception is in the code required to set VGA mode X. Since mode X is not standard, it must be set by programming the VGA registers directly. In this case the code has to access registers in the CRT Controller group in order to expand the vertical scanning range. These operations are shown in the SET_MODEX procedure listed in Chapter 6.

The CRT Controller registers related to cursor size and position are also occasionally programmed directly. Since programs that execute in a VGA graphics mode have no access to the hardware cursor, these registers are not considered in this book.

Another group of registers within the CRT Controller that are occasionally programmed directly are those that determine the start address of the screen window in the video buffer. This manipulation is sometimes used in scrolling and panning text and in graphics mode manipulations. In VGA systems the CRT Controller Start Address High and Start Address Low registers (offset 0CH and 0DH) locate the screen window within a byte offset, while the Preset Row Scan register (offset 08H) locates the window at the closest pixel row. Therefore the Preset Row Scan register is used to determine the vertical pixel offset of the screen window. The horizontal pixel offset of the screen window is programmed by changing the value stored in the Horizontal Pixel Pan register of the Attribute Controller, described later in this chapter. Figure 5.7 shows the Start Address registers of the CRT Controller as well as the Preset Row Scan register.

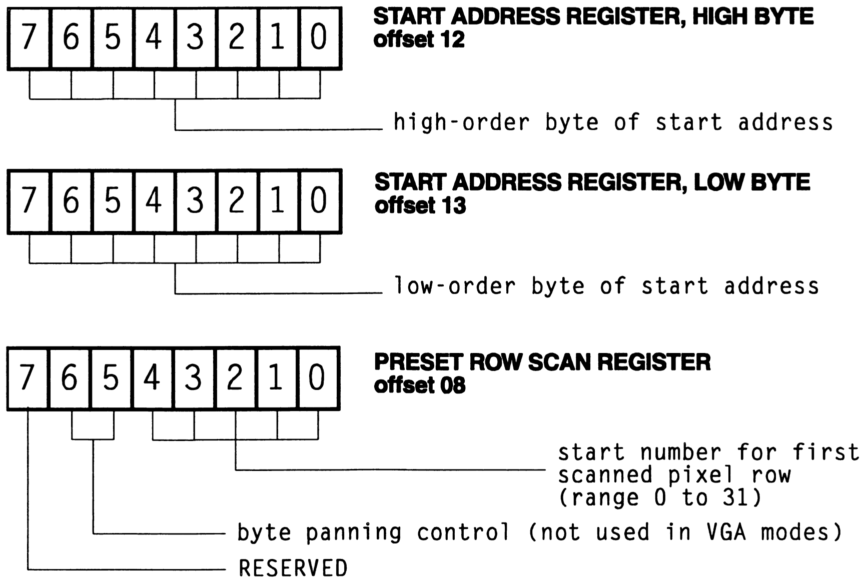

7 6 5 4 3 2 1 0
START ADDRESS REGISTER, HIGH BYTE
offset 12
high-order byte of start address
7 6 5 4 3 2 1 0
START ADDRESS REGISTER, LOW BYTE
offset 13
low-order byte of start address
7 6 5 4 3 2 1 0
PRESET ROW SCAN REGISTER
offset 08
start number for first scanned pixel row (range 0 to 31)
byte panning control (not used in VGA modes)
RESERVED

Figure 5.7 *Start Address and Preset Row Scan Registers*

## 5.2.3 VGA Sequencer

The VGA Sequencer register group controls memory fetch operations and provides timing signals for the dynamic RAMs. This allows the microprocessor to access video memory in cycles inserted between the display memory cycles. Table 5.4 shows the registers in the VGA Sequencer.

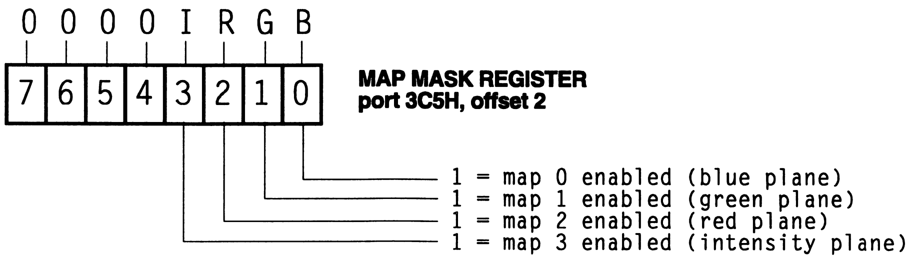
Table 5.4 *The VGA Sequencer Registers*

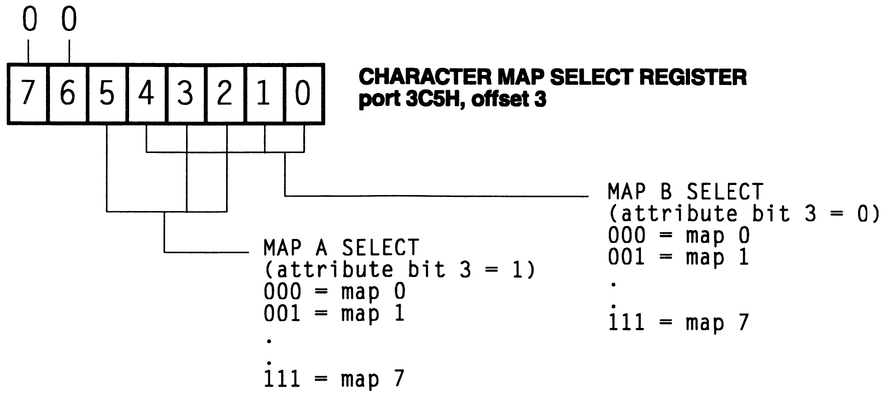
| PORT | OFFSET | DESCRIPTION |
|---|---|---|
| 03C4H | | Address register |
| 03C5H | 0 | Synchronous or Asynchronous Reset |
| | 1 | Clocking Mode |
| | 2 | Map Mask |
| | 3 | Character Map Select |
| | 4 | Memory Mode |

The Address register of the Sequencer group is used to select which one of the Data registers is currently accessed. Only the 3 low-order bits of the Address register are used. The Data register at offset 0 (see Table 5.4) is used during system reset. The Clocking Mode register is also used mostly during mode setting, except for bit 5, which can be used to turn off the display. Turning off the display assigns all memory access time to the CPU, which can be used to perform a rapid full screen update. The most used registers of the Sequencer group are the Map Mask, Character Map Select, and Memory Mode. Figure 5.8 is a bitmap of these registers.

The Map Mask register in the Sequencer group allows the protection of any specific memory map by masking it from the microprocessor and from the Character Map Select register. If VGA is in a color graphics mode, the Map Mask can be used to select the color at which one or more pixels are displayed. The color is encoded in the IRGB format, as shown in Figure 5.8. To access the Map Mask register, first load the value 2 into the address register of the Sequencer, at port 3C4H, which is the offset of the Map Mask register. The following code fragment shows the usual program operations regarding the Map Mask register:

```
; Setting 8 bright-red pixels in VGA mode number 18
; The code assumes that video mode number 18 is selected,
; that ES is set to the video segment base, and that BX points
; to the offset of the first pixel to be set
;***********************|
;    select register    |
;***********************|
        MOV     DX,3C4H         ; Address register of Sequencer
        MOV     AL,2            ; Offset of Map Mask
        OUT     DX,AL           ; Map Mask selected
        MOV     DX,3C5H         ; Data to Map Mask
```


0 0 0 0 I R G B
7 6 5 4 3 2 1 0
MAP MASK REGISTER
port 3C5H, offset 2
1 = map 0 enabled (blue plane)
1 = map 1 enabled (green plane)
1 = map 2 enabled (red plane)
1 = map 3 enabled (intensity plane)



LOCATION OF MAP TABLES

| map No. | location | map No. | location |
|---|---|---|---|
| 0 | 1st 8K of map 2 | 4 | 2nd 8K of map 2 |
| 1 | 3rd 8K of map 2 | 5 | 4th 8K of map 2 |
| 2 | 5th 8K of map 2 | 6 | 6th 8K of map 2 |
| 3 | 7th 8K of map 2 | 7 | 8th 8K of map 2 |

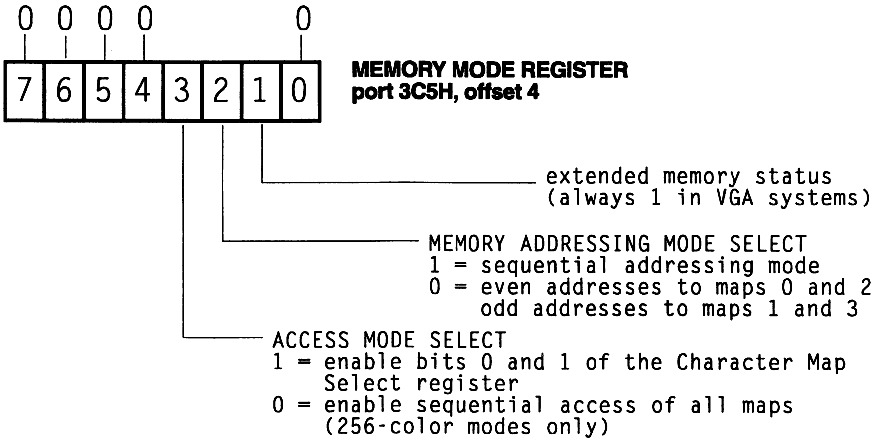

0 0 0 0 0
7 6 5 4 3 2 1 0
MEMORY MODE REGISTER
port 3C5H, offset 4
extended memory status
(always 1 in VGA systems)
MEMORY ADDRESSING MODE SELECT
1 = sequential addressing mode
0 = even addresses to maps 0 and 2
odd addresses to maps 1 and 3
ACCESS MODE SELECT
1 = enable bits 0 and 1 of the Character Map Select register
0 = enable sequential access of all maps
(256-color modes only)

Figure 5.8 *Registers in the Sequencer Group*

```
        MOV     AL,00001100B    ; Intensity and red bits set
                                ; in IRGB encoding
        OUT     DX,AL           ; Map Mask = 0000 IR00
;***********************|
;       set pixels      |
;***********************|
; Setting the pixels consists of writing a 1 bit in the
; corresponding buffer address.
        MOV     AL,ES:[BX]      ; Dummy read operation
        MOV     AL,11111111B    ; Set all bits
        MOV     ES:[BX],AL      ; Write to video buffer
;
;***********************|
;   restore Map Mask    |
;***********************|
; Restore the Map Mask to the default state
        MOV     DX,3C4H         ; Address register of Sequencer
        MOV     AL,02H          ; Offset of Map Mask
        OUT     DX,AL           ; Map Mask selected
        MOV     DX,3C5H         ; Data to Map Mask
        MOV     AL,00001111B    ; Default IRGB code for Map Mask
        OUT     DX,AL           ; Map Mask = 0000 IRGB
        .
        .
        .
```

The Character Map Select register of the Sequencer is used in selecting one of the BIOS character maps. This operation is related to reprogramming bit 3 of the attribute byte, as mentioned in Section 5.1.1. In this case bit 3 serves to select one of two character sets. Normally the character maps, named A and B, have the same value and bit 3 of the attribute byte is used to control the bright or normal display of the character foreground. When the Character Map Select register is programmed so that character maps A and B have different values, then bit 3 of the attribute byte is used to toggle between two sets of 256 characters each. Since the use of multiple VGA character sets is used mostly when programming in VGA alphanumeric modes, this matter is not discussed further.

The Memory Mode register of the Sequencer is related to the display modes. Most programs leave the setting of this register to the BIOS mode select services.

### 5.2.4 VGA Graphics Controller

The registers in the Graphics Controller group serve to interface video memory with the Attribute Controller and with the system microprocessor. The Graphic Controller is bypassed in the alphanumeric modes. Table 5.5 lists the registers in the VGA Graphics Controller group. All the registers in the Graphics Controller are of interest to the graphics and animation programmer.

Table 5.5 *The VGA Graphics Controller Registers*

| PORT | OFFSET | DESCRIPTION |
|---|---|---|
| 03CEH | | Address register |
| 03CFH | 0 | Set/Reset |
| | 1 | Enable Set/Reset |
| | 2 | Color Compare for read mode 1 operation |
| | 3 | Logical Operation Select and Data Rotate |
| | 4 | Read Operation Map Select |
| | 5 | Select Graphics Mode |
| | 6 | Miscellaneous Operations |
| | 7 | Read Mode 1 Color Don't Care |
| | 8 | Bit Mask |

Figure 5.9 shows the bitmaps for six registers in the Graphics Controller group.

The Set/Reset register is used to permanently set or clear a specific bit plane. This operation can be useful in writing a specific color to the entire screen or in disabling a color map. The Set/Reset register affects only write mode 0 operations. The use of the Set/Reset register requires the use of the Enable Set/Reset register. Enable Set/Reset determines which of the maps is accessed by the Set/Reset register. This mechanism provides a double-level control over the four maps.

The Color Compare register is used during read mode 1 operations of some VGA modes to test for the presence of memory bits that match one or more color maps. For example, if a program sets bit 0 (blue) and bit 3 (intensity) of the Color Compare register, a subsequent memory read operation shows a 1 value for those pixels whose intensity and blue maps are set, while all other combinations are reported with a zero value. One or more bit planes can be excluded from the compare by clearing the corresponding bit in the Color Don't Care register. If the intensity bit is zero in the Color Don't Care register, a color compare operation for the blue bit map is positive for all pixels in blue or bright blue color.

The Data Rotate register determines how data is combined with data latched in the system microprocessor registers. The possible logical operations are AND, OR, and XOR. If bits 3 and 4 are reset, data is unmodified. A second function of this register is to right-rotate data from 0 to 7 places. This function is controlled by bits 0 to 2.

VGA video memory in the graphics modes is based on encoding the color of a single pixel into several memory maps. The Read Map Select register is used to determine which map is read by the system microprocessor. The following code fragment shows the use of the Read Operation Map Select register:

```
; Code to read the contents of the 4 color maps in VGA mode 18
; Code assumes that read mode 0 has been previously set
; On entry:
;               ES = A000H
;               BX = byte offset into video map
```

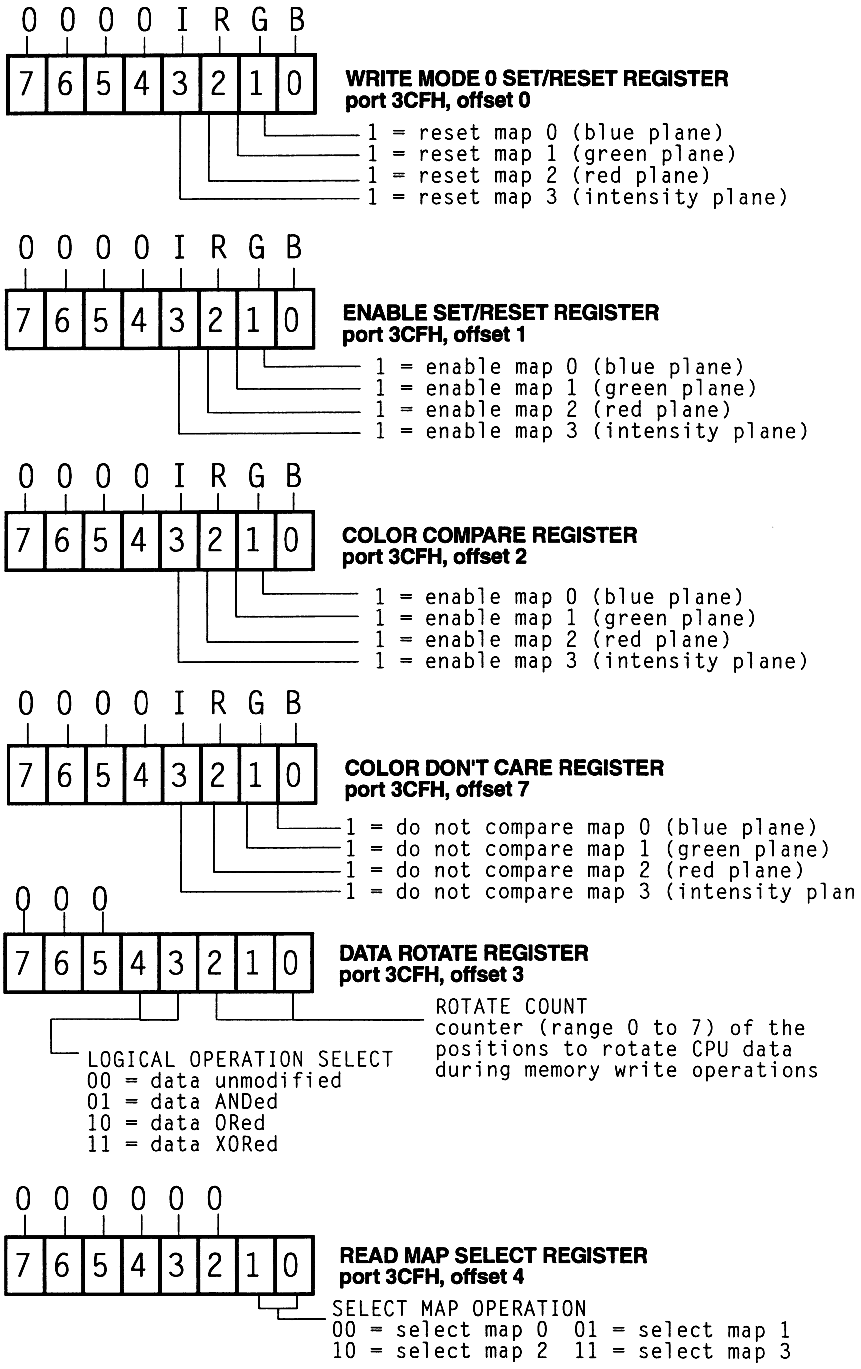

0 0 0 0 I R G B
7 6 5 4 3 2 1 0
WRITE MODE 0 SET/RESET REGISTER
port 3CFH, offset 0
1 = reset map 0 (blue plane)
1 = reset map 1 (green plane)
1 = reset map 2 (red plane)
1 = reset map 3 (intensity plane)
0 0 0 0 I R G B
7 6 5 4 3 2 1 0
ENABLE SET/RESET REGISTER
port 3CFH, offset 1
1 = enable map 0 (blue plane)
1 = enable map 1 (green plane)
1 = enable map 2 (red plane)
1 = enable map 3 (intensity plane)
0 0 0 0 I R G B
7 6 5 4 3 2 1 0
COLOR COMPARE REGISTER
port 3CFH, offset 2
1 = enable map 0 (blue plane)
1 = enable map 1 (green plane)
1 = enable map 2 (red plane)
1 = enable map 3 (intensity plane)
0 0 0 0 I R G B
7 6 5 4 3 2 1 0
COLOR DON'T CARE REGISTER
port 3CFH, offset 7
1 = do not compare map 0 (blue plane)
1 = do not compare map 1 (green plane)
1 = do not compare map 2 (red plane)
1 = do not compare map 3 (intensity plan
0 0 0
7 6 5 4 3 2 1 0
DATA ROTATE REGISTER
port 3CFH, offset 3
ROTATE COUNT
counter (range 0 to 7) of the
positions to rotate CPU data
during memory write operations
LOGICAL OPERATION SELECT
00 = data unmodified
01 = data ANDed
10 = data ORed
11 = data XORed
0 0 0 0 0 0
7 6 5 4 3 2 1 0
READ MAP SELECT REGISTER
port 3CFH, offset 4
SELECT MAP OPERATION
00 = select map 0 01 = select map 1
10 = select map 2 11 = select map 3

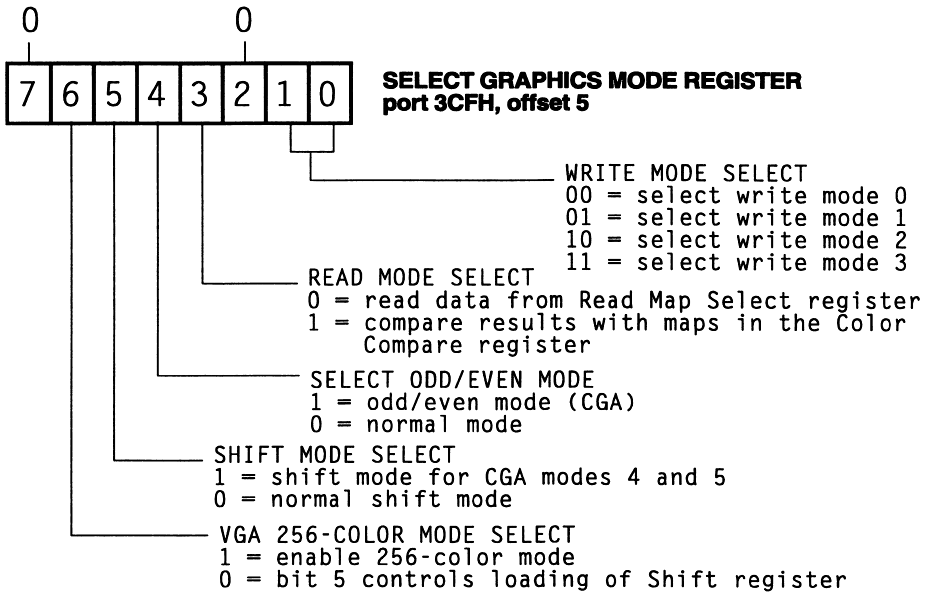
Figure 5.9 *Registers in the Graphics Controller Group*

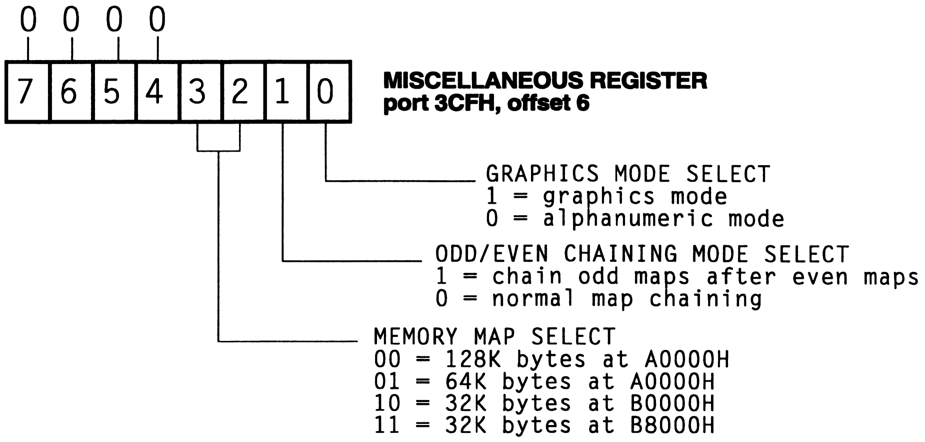
```
; On exit:
;               CL = byte stored in intensity map
;               CH = byte stored in red map
;               DL = byte stored in green map
;               DH = byte stored in blue map
;
; Set counter and map selector
        MOV     CX,4            ; Counter for 4 maps to read
        MOV     DI,0            ; Map selector code
;
READ_IRGB:
; Select map from which to read
        MOV     DX,3CEH         ; Graphic Controller Address
                                ; register
        MOV     AL,4            ; Read Operation Map Select
        OUT     DX,AL           ; register
;
        INC     DX              ; Graphic controller at 3CFH
        MOV     AX,DI           ; AL = map selector code (in DI)
        OUT     DX,AL           ; IRGB color map selected
; Read 8 bits from selected map
        MOV     AL,ES:[BX]      ; Get byte from bit plane
        PUSH    AX              ; Store it in the stack
        INC     DI              ; Bump selector to next map
        LOOP    READ_IRGB       ; Execute loop 4 times
;
; 4 maps are stored in stack
; Retrieve maps into exit registers
        POP     AX              ; B map byte in AL
        MOV     DH,AL           ; Move B map byte to DH
        POP     AX              ; G map byte in AL
        MOV     DL,AL           ; Move G map byte to DL
        POP     AX              ; R map byte in AL
        MOV     CH,AL           ; Move R map byte to CH
        POP     AX              ; I map byte in AL
        MOV     CL,AL           ; Move I map byte to CL
        .
        .
        .
```

The Select Graphics Mode register, the Miscellaneous register, and the Bit Mask register of the Graphics Controller group are shown in Figure 5.10.

## VGA Read and Write Modes

VGA systems allow several ways for performing memory read and write operations, usually known as the read and write modes. The Select Graphics Mode register of the Graphics Controller group allows the programmer to select which of two read and four write modes is active. The four VGA write modes can be described as follows:


0
0
7 6 5 4 3 2 1 0
SELECT GRAPHICS MODE REGISTER
port 3CFH, offset 5
WRITE MODE SELECT
00 = select write mode 0
01 = select write mode 1
10 = select write mode 2
11 = select write mode 3
READ MODE SELECT
0 = read data from Read Map Select register
1 = compare results with maps in the Color Compare register
SELECT ODD/EVEN MODE
1 = odd/even mode (CGA)
0 = normal mode
SHIFT MODE SELECT
1 = shift mode for CGA modes 4 and 5
0 = normal shift mode
VGA 256-COLOR MODE SELECT
1 = enable 256-color mode
0 = bit 5 controls loading of Shift register



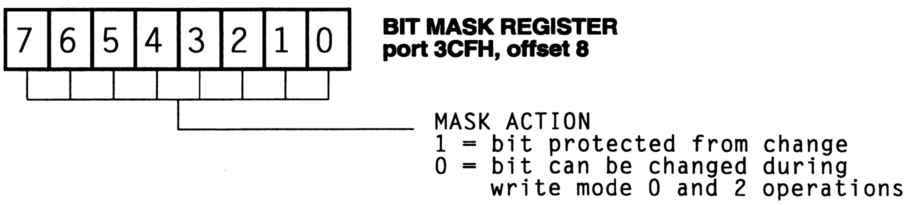

7 6 5 4 3 2 1 0
BIT MASK REGISTER
port 3CFH, offset 8
MASK ACTION
1 = bit protected from change
0 = bit can be changed during write mode 0 and 2 operations

Figure 5.10 *Other Registers in the Graphics Controller Group*

*Write mode 0* is the default write mode. In this write mode, the Map Mask register of the Sequencer group, the Bit Mask register of the Graphics Controller group, and the CPU are used to set the screen pixel to a desired color.

In *write mode 1* the contents of the latch registers are first loaded by performing a read operation and then copied directly onto the color maps by performing a write operation. This mode is often used in moving areas of memory.

*Write mode 2*, a simplified version of write mode 0, also allows setting an individual pixel to any desired color. However, in write mode 2 the color code is contained in the CPU byte.

In *write mode 3* the byte in the CPU is ANDed with the contents of the Bit Mask register of the Graphic Controller.

The write mode is selected by setting bits 0 and 1 of the Graphic Controller's Graphic Mode register. It is a good programming practice to preserve the remaining bits in this register when modifying bits 0 and 1. This is performed by reading the Graphic Mode register, altering the write mode bits, and then resetting the register without changing the remaining bits. The following code fragment sets a write mode in a VGA system. The remaining bits in the Select Graphics Mode register are preserved.

```
; Set the Graphics Controller's Select Graphic Mode register
; to the write mode in the AH register
        MOV     DX,3CEH         ; Graphic Controller Address
                                ; register
        MOV     AL,5            ; Offset of Mode register
        OUT     DX,AL           ; Select this register
        INC     DX              ; Point to Data register
        IN      AL,DX           ; Read register contents
        AND     AL,11111100B    ; Clear bits 0 and 1
        OR      AL,AH           ; Set mode in AL low bits
        MOV     DX,3CEH         ; Address register
        MOV     AL,5            ; Offset of Mode Register
        OUT     DX,AL           ; Select again
        INC     DX              ; Point to Data register
        OUT     DX,AL           ; Output to Mode Register
; Note: the Select Mode register is read-only in EGA systems
;       and therefore this code will not work correctly
```

Note that bit 6 of the Graphics Mode register must be set for 256-color modes and cleared for the remaining ones. The SET_WRITE_256 procedure listed in Chapter 6 sets write mode 0 and the 256-color bit so that VGA mode 19, in 256 colors, operates correctly.

Once a write mode is selected, the program can access video memory to set the desired screen pixels, as in the following code fragment:

```
; Write mode 2 pixel setting routine
; On entry:
;              ES = A000H
;              BX = byte offset into the video buffer
;              AL = pixel color in IRGB format
;              AH = bit pattern to set (mask)
;
; Note: this procedure does not reset the default read or write
; modes or the contents of the Bit Mask register.
; The code assumes that write mode 2 has been set previously
        PUSH    AX          ; Color byte
        PUSH    AX          ; Twice
;**********************|
;     set bit mask     |
;**********************|
; Set Bit Mask register according to value in AH
        MOV     DX,3CEH     ; Graphic controller address
        MOV     AL,8        ; Offset = 8
        OUT     DX,AL       ; Select Bit Mask register
        INC     DX          ; To 3CFH
        POP     AX          ; Color code once from stack
        MOV     AL,AH       ; Bit pattern
        OUT     DX,AL       ; Load bit mask
;**********************|
;   write color        |
;**********************|
        MOV     AL,ES:[BX]  ; Dummy read to load latch
                            ; registers
        POP     AX          ; Restore color code
        MOV     ES:[BX],AL  ; Write the pixel with the
                            ; color code in AL
        .
        .
        .
```

The VGA also provides two read modes. In read mode 0, the default read mode, the CPU is loaded with the contents of one of the color maps. In read mode 1, the contents of the maps are compared with a predetermined value before being loaded into the CPU. The active read mode depends on the setting of bit 3 of the Graphic Mode Select register, in the Graphics Controller. The SET_READ_MODE procedure listed in Chapter 6 performs this operation.

The Miscellaneous register is used in conjunction with the Select Graphics Modes register to enable specific graphics function. Bits 2 and 3 of the Miscellaneous register control the mapping of the video buffer in the system's memory space. The normal mapping of each mode can be seen in the buffer address column of Table 5.1. The manipulation of the Miscellaneous register is usually left to the BIOS mode change service.

All read and write operations performed by the VGA take place at a byte level. However, in certain graphics modes, such as mode 18, video data is stored at a bit level in four color maps. In this case, the code must mask out the undesired color maps in order to determine the state of an individual screen pixel or to set a pixel to a certain color. The TEST instruction provides a convenient way for determining an individual screen pixel following a read operation. The Bit Mask register permits setting individual pixels while in write modes 0 and 2.

In the execution of write operations while in VGA mode 18, the bit mask for setting an individual screen pixel can be found from a look-up table or by right-shifting a unitary bit pattern (10000000B). The following code fragment calculates the offset into the video buffer and the bit mask required for writing an individual pixel using VGA write modes 0 or 2.

```
; Mask and offset computation from x and y pixel coordinates
; Code is for VGA mode number 18 (640 by 480 pixels)
;
; On entry:
;              CX = x coordinate of pixel (range 0 to 639)
;              DX = y coordinate of pixel (range 0 to 479)
;
; On exit:
;              BX = byte offset into video buffer
;              AH = bit mask for the write operation using
;                   write modes 0 or 2
;
;**********************|
;  calculate address   |
;**********************|
        PUSH    AX              ; Save accumulator
        PUSH    CX              ; Save x coordinate
        MOV     AX,DX           ; y coordinate to AX
        MOV     CX,80           ; Multiplier (80 bytes per row)
        MUL     CX              ; AX = y times 80
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
; Prepare for division
        MOV     CL,8            ; Load divisor
        DIV     CL              ; AX / CL = quotient in AL and
                                ; remainder in AH
; Add in quotient
        MOV     CL,AH           ; Save remainder in CL
        MOV     AH,0            ; Clear high byte
        ADD     BX,AX           ; Offset into buffer to BX
        POP     AX              ; Restore AX
; Compute bit mask from remainder
        MOV     AH,10000000B        ; Unitary mask for 0 remainder
        SHR     AH,CL               ; Shift right CL times
; The byte offset (in BX) and the pixel mask (in AH) can now
; be used to set the individual screen pixel
        .
        .
        .
```

### 5.2.5 VGA Attribute Controller

The Attribute Controller receives color data from the Graphics Controller and formats it for the video display hardware. Input to the Attribute Controller is in the form of attribute data in the alphanumeric modes and in the form of serialized bit plane data in the graphics modes. The data is converted into 8-bit digital color output to the DAC. Blinking, underlining, and cursor display logic are also controlled by this register. In VGA systems the output of the Attribute Controller goes directly to the video DAC and the CRT. Table 5.6 shows the registers in the Attribute Controller group.

Table 5.6 *The VGA Attribute Controller Registers*

| PORT | OFFSET | DESCRIPTION |
|---|---|---|
| 3C0H | | Attribute Address and Palette Address register |
| 3C1H | | Read operations |
| 3C0H | 0–15 | Palette registers |
| | 16 | Attribute Mode Control |
| | 17 | Screen Border Color Control (overscan color) |
| | 18 | Color Plane Enable |
| | 19 | Horizontal Pixel Panning |
| | 20 | Color Select |

Register addressing in the Attribute Controller group is performed differently than with the other VGA registers. This is due to the fact that the Attribute Controller does not have a dedicated bit to control the selection of its internal address and data registers. Instead, the Attribute Controller uses an internal flip-flop to toggle the address and data functions. This explains why the Index and the Data registers of the Attribute Controller are both mapped to port 3C0H (see Table 5.6). Figure 5.11 shows the Attribute and Palette Address registers, the Palette Address register, and the Attribute Mode Control register in the Attribute Controller group.

Programming the Attribute Controller requires accessing Input Status register 1 of the General register (see Figure 5.6) in order to clear the flip-flop. The address of the Status register 1 is 3BAH in monochrome modes and 3DAH in color modes. The complete sequence of operations for writing data to the Attribute Controller is as follows:

1. Issue an IN instruction to address 3BAH (in color modes) or to address 3DAH (in monochrome modes) to clear the flip-flop and select the address function of the Attribute Controller.
2. Disable interrupts.
3. Issue an OUT instruction to the address register, at port 3C0H, with the number of the desired data register.
4. Issue another OUT instruction to this same port to load a value into the Data register.
5. Enable interrupts.

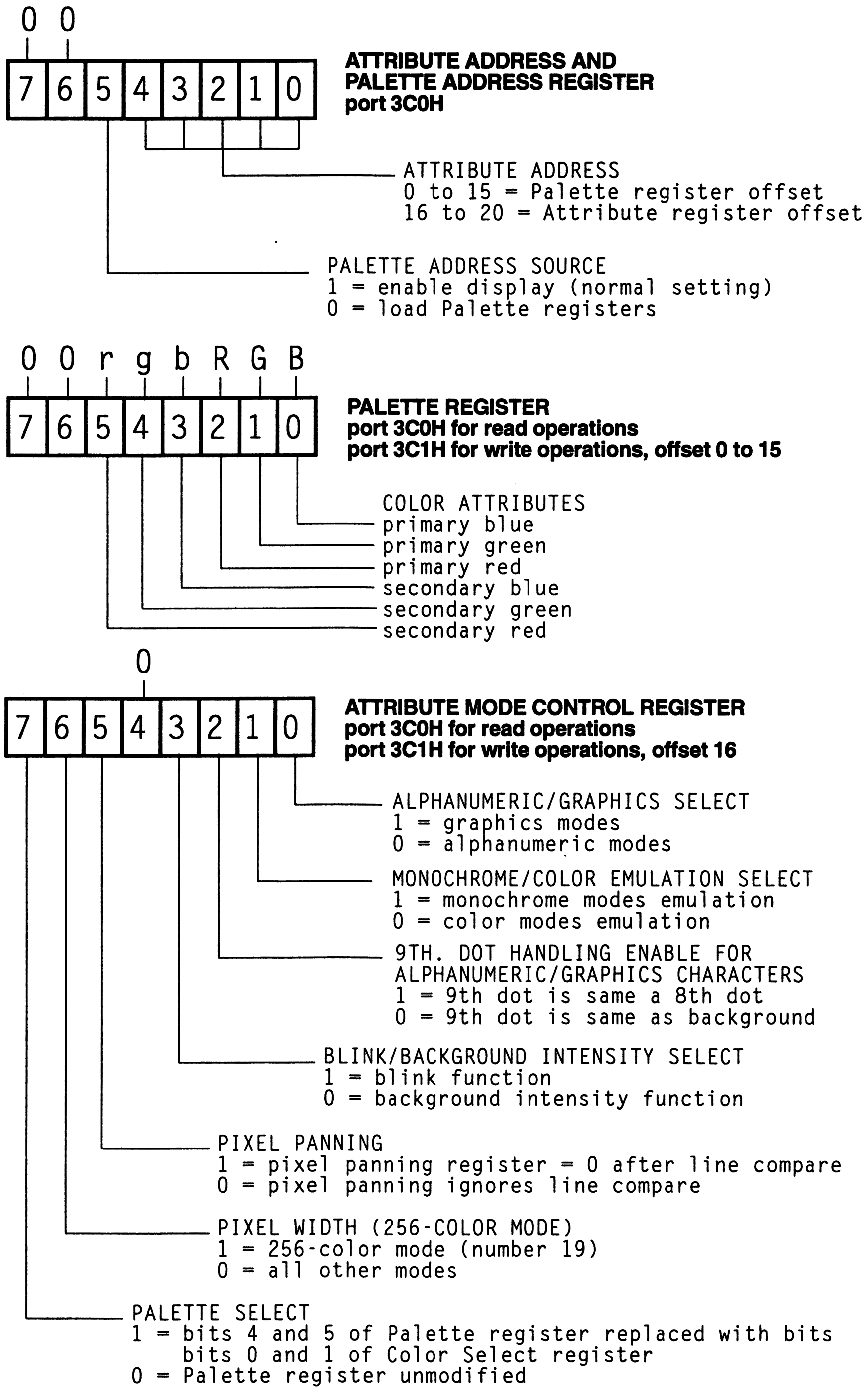

0 0
7 6 5 4 3 2 1 0
ATTRIBUTE ADDRESS AND PALETTE ADDRESS REGISTER port 3C0H
ATTRIBUTE ADDRESS
0 to 15 = Palette register offset
16 to 20 = Attribute register offset
PALETTE ADDRESS SOURCE
1 = enable display (normal setting)
0 = load Palette registers
0 0 r g b R G B
7 6 5 4 3 2 1 0
PALETTE REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 0 to 15
COLOR ATTRIBUTES
primary blue
primary green
primary red
secondary blue
secondary green
secondary red
0
7 6 5 4 3 2 1 0
ATTRIBUTE MODE CONTROL REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 16
ALPHANUMERIC/GRAPHICS SELECT
1 = graphics modes
0 = alphanumeric modes
MONOCHROME/COLOR EMULATION SELECT
1 = monochrome modes emulation
0 = color modes emulation
9TH. DOT HANDLING ENABLE FOR ALPHANUMERIC/GRAPHICS CHARACTERS
1 = 9th dot is same a 8th dot
0 = 9th dot is same as background
BLINK/BACKGROUND INTENSITY SELECT
1 = blink function
0 = background intensity function
PIXEL PANNING
1 = pixel panning register = 0 after line compare
0 = pixel panning ignores line compare
PIXEL WIDTH (256-COLOR MODE)
1 = 256-color mode (number 19)
0 = all other modes
PALETTE SELECT
1 = bits 4 and 5 of Palette register replaced with bits bits 0 and 1 of Color Select register
0 = Palette register unmodified

Figure 5.11 *Registers in the Attribute Controller Group*

In 16-color modes, the 16 Palette registers of the Attribute Controller determine how the 16 color values in the IRGB bit planes are displayed. The default values for the Palette registers are shown in Table 5.7.

Table 5.7 *Default Setting of VGA/EGA Palette Registers*

| REGISTER OFFSET | HEX | BITS 0–5 rgbRGB | COLOR |
|---|---|---|---|
| 00H | 00H | 0 0 0 0 0 0 | Black |
| 01H | 01H | 0 0 0 0 0 1 | Blue |
| 02H | 02H | 0 0 0 0 1 0 | Green |
| 03H | 03H | 0 0 0 0 1 1 | Cyan |
| 04H | 04H | 0 0 0 1 0 0 | Red |
| 05H | 05H | 0 0 0 1 0 1 | Magenta |
| 06H | 14H | 0 1 0 1 0 0 | Brown |
| 07H | 07H | 0 0 0 1 1 1 | White |
| 08H | 38H | 1 1 1 0 0 0 | Dark grey |
| 09H | 39H | 1 1 1 0 0 1 | Light blue |
| 0AH | 3AH | 1 1 1 0 1 0 | Light green |
| 0BH | 3BH | 1 1 1 0 1 1 | Light cyan |
| 0CH | 3CH | 1 1 1 1 0 0 | Light red |
| 0DH | 3DH | 1 1 1 1 0 1 | Light magenta |
| 0EH | 3EH | 1 1 1 1 1 0 | Yellow |
| 0FH | 3FH | 1 1 1 1 1 1 | Intensified white |

Each VGA Palette register consists of six bits, allowing 64 color combinations in each register. The bits labeled "RGB" in Table 5.7 correspond to the primary values for red, green and blue colors, and the bits labeled "rgb" correspond to the secondary values. Since each color is represented by two bits, each one can have four possible levels of saturation; for example, the levels of saturation for red are:

| Saturation | rgbRGB | Interpretation |
|---|---|---|
| 0 | 000000 | no red |
| 1 | 100000 | low red |
| 2 | 000100 | red |
| 3 | 100100 | high red |

The Palette registers can be changed by means of BIOS service number 16, interrupt 10H, or by programming the Attribute Controller registers directly. Note that the setting of the Palette registers does not affect the color output in the 256-color modes since, in this case, the 8-bit color values in video memory are transmitted directly to the DAC.

The Attribute Mode Control register serves to select the characteristics associated with the video mode. Bit 0 selects whether the display is in an alphanumeric or in a graphics mode. Bit 1 determines if VGA operates in a monochrome or color emulation. Bit 2 is related to the handling of the ninth screen dot while displaying the graphics characters in the range C0H to DFH. If this bit is set, the graphics characters in this range generate unbroken horizontal lines. This feature refers to the MDA emulation mode only, since other character fonts do not have the ninth dot. BIOS sets this bit automatically in the modes that require it.

Bit 5 of the Attribute Mode Control register in the Attribute Controller group relates to independently panning the screen sections during split-screen operation. Bit 6 of the Attribute Mode Control register is set to 1 during operation in mode number 19 (256 colors) and cleared for all other modes. Finally, bit 7 of the Attribute Mode Control register determines the source for the bits labeled r and g (numbers 4 and 5) in the Palette register. If bit 7 is set, the r and g bits in the Palette register are replaced by bits 0 and 1 of the Color Select register. If bit 7 is reset, then all Palette register bits are sent to the DAC.

Figure 5.12 shows the bitmaps of the Overscan Color register, the Color Plane Enable register, the Horizontal Pixel Panning register, and the Color Select register of the Attribute Controller group.

In some alphanumeric and graphics modes the VGA display area is surrounded by a colored band. The width of this band is the same as the width of a single character (8 pixels) in the 80-column modes. The color of this border area is determined by the Overscan Color register of the Attribute Controller. Normally the screen border is not noticeable, since the default border color is black. The border color is not available in the 40-column alphanumeric modes or in the graphics modes with 320 pixel rows, except for VGA graphics mode number 19.

The Color Plane Enable register allows excluding one or more bit planes from the color generation process. The main purpose of this function is to provide compatibility with EGA systems equipped with less than 256K of memory. Bits 4 and 5 of this register are used in system diagnostics.

The Horizontal Pixel Panning register of the Attribute Controller is used to shift video data horizontally to the left, pixel by pixel. This feature is available in the alphanumeric and graphics modes. The number of pixels that can be shifted is determined by the display mode. In the VGA 256-color graphics mode the maximum number of allowed pixels is three. In alphanumeric modes 0, 1, 2, 3, and 7, the maximum is eight pixels. In all other modes the maximum is seven pixels. The Horizontal Pixel Panning register can be programmed in conjunction with the Video Buffer Start Address registers of the CRT Controller to implement smooth horizontal screen scrolling in alphanumeric and in graphics modes. These manipulations are described in Chapter 6.

The Color Select register of the Attribute Controller provides additional color selection flexibility to the VGA system, as well as a way for rapidly switching between sets of displayed colors. When bit 7 of the Attribute Mode Control register is clear, the 8-bit color value sent to the DAC is formed by the six bits

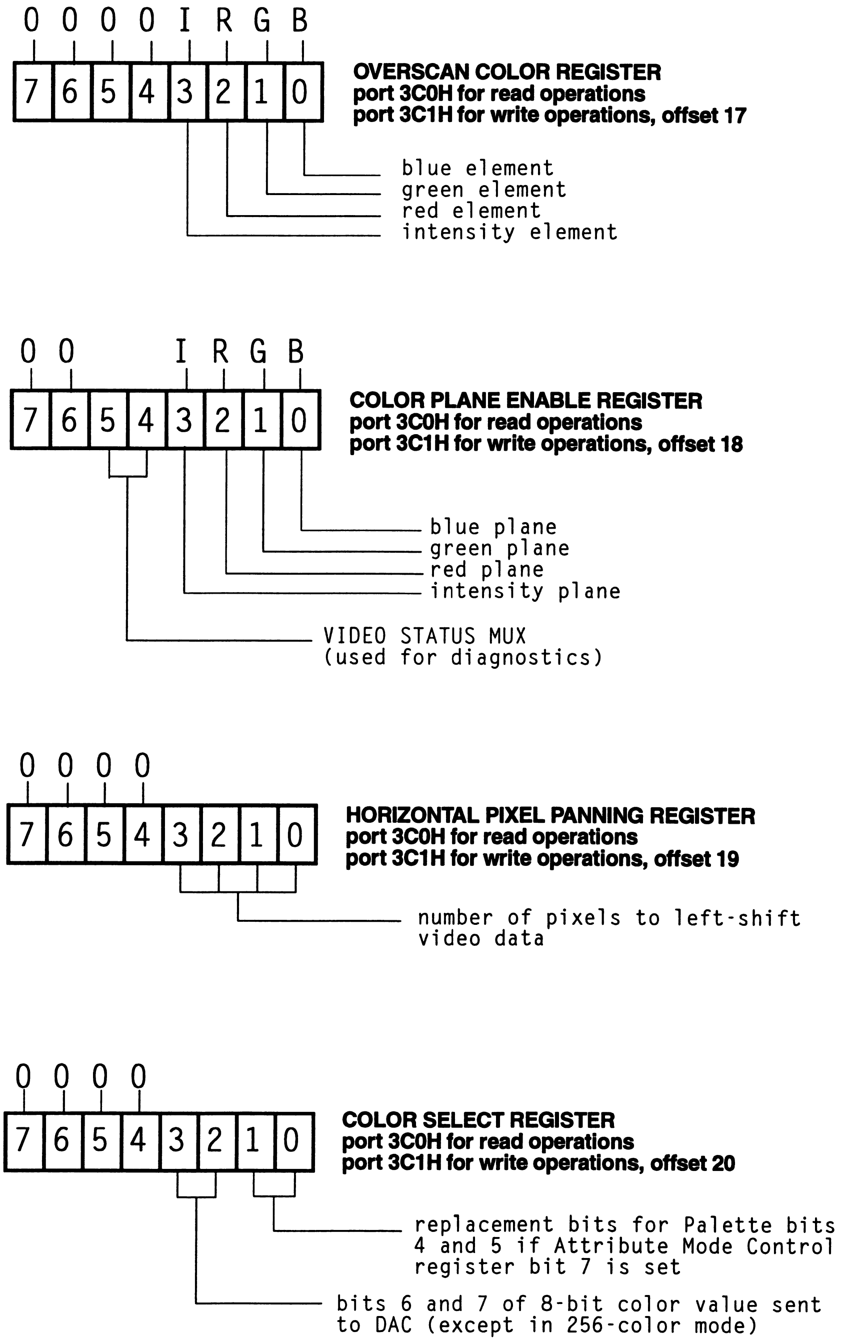

0 0 0 0 I R G B
7 6 5 4 3 2 1 0
OVERSCAN COLOR REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 17
blue element
green element
red element
intensity element
0 0 I R G B
7 6 5 4 3 2 1 0
COLOR PLANE ENABLE REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 18
blue plane
green plane
red plane
intensity plane
VIDEO STATUS MUX
(used for diagnostics)
0 0 0 0
7 6 5 4 3 2 1 0
HORIZONTAL PIXEL PANNING REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 19
number of pixels to left-shift video data
0 0 0 0
7 6 5 4 3 2 1 0
COLOR SELECT REGISTER
port 3C0H for read operations
port 3C1H for write operations, offset 20
replacement bits for Palette bits 4 and 5 if Attribute Mode Control register bit 7 is set
bits 6 and 7 of 8-bit color value sent to DAC (except in 256-color mode)

Figure 5.12 *Other Registers in the Attribute Controller Group*

from the Palette registers and bits 2 and 3 of the Color Select register. If bit 7 of the Attribute Mode Control register is set, then the 8-bit color value is formed with the lower four bits of the Palette register and the four bits of the Color Select register. Since these bits affect all Palette registers simultaneously, the program can rapidly change all displayed colors by changing the value in the Color Select register. The Color Select register is not used in the 256-color graphics mode 19.

## 5.3 VGA Digital-to-Analog Converter (DAC)

The Digital-to-Analog Converter, or DAC, provides a set of 256 color registers, sometimes called the color look-up table, as well as three color drivers for an analog display. The DAC register set permits displaying 256 color combinations from a total of 262,144 possible colors. Table 5.8 shows the DAC registers.

Table 5.8 *VGA Video Digital-to-Analog Converter Addresses*

| REGISTER | OPERATIONS | ADDRESS |
|---|---|---|
| Pixel Address (write operations) | Read-Write | 3C8H |
| Pixel Address (read operations) | Write only | 3C7H |
| DAC State register | Read only | 3C7H |
| Pixel Data register | Read-Write | 3C9H |
| Pixel Mask register | Read-Write | 3C6H |

**Note:** Applications must not write to the Pixel Mask register or the color look-up table could be destroyed.

Each of the DAC's 256 registers uses six data bits to encode the value of the primary colors red, green, and blue. This design determines that each DAC register is 18 bits wide. It is the possible combinations of 18 bits that allow 262,144 DAC colors. Note that the VGA color registers in the DAC duplicate the color control offered by the Palette registers of the Attribute Controller. In fact, the VGA Palette registers are provided for compatibility with the EGA card, which does not contain DAC registers. When compatibility with the EGA is not an issue, VGA programming can be simplified by ignoring the Palette registers and making all color manipulations in the DAC. Furthermore, the Palette registers are disabled when VGA is in the 256-color mode 19, since mode 19 has no EGA equivalent.

Figure 5.13 shows the bitmaps of the Pixel Address and DAC State registers. The DAC Pixel Address register holds the number (also called the address) of one of the 256 DAC registers. Read operations to the Pixel Address register are performed to port 3C7H and write operations to port 3C8H (see Table 5.8). A write operation changes the 18-bit color stored in the register (in red/green/blue format). A read operation is used to obtain the RGB value currently stored in the DAC register.

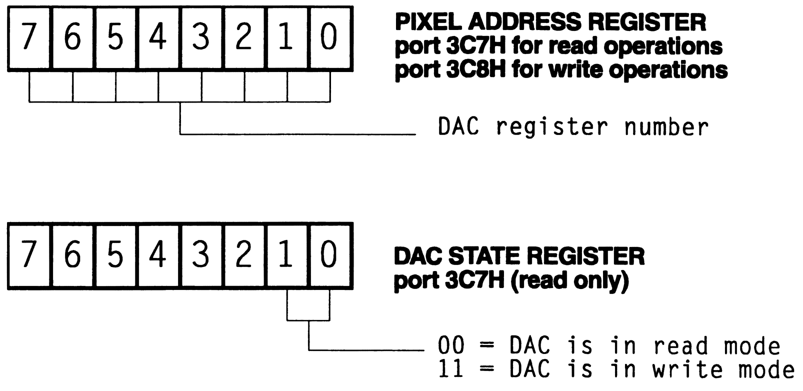


Figure 5.13 *Registers in the VGA DAC*

The DAC State register encodes whether the DAC is in read or write mode. A mode change takes place when the Pixel Address register is accessed: if the Pixel Address register is set at port 3C7H, then the DAC goes into a read mode, if it is set at port 3C8H, then the DAC goes into a write mode. Notice that although the Pixel Address register for read operations and the DAC State register are both mapped to port 3C7H there is no occasion for conflict, since the DAC State register is read only and the Pixel Address register for read operations is write only.

The Pixel Data register in the DAC is used to hold three six-bit data items representing a color value in RGB format. The Pixel Data register can be read after the program has selected the corresponding DAC register at the Pixel Address read operation port 3C7H. The Pixel Data register can be written after the program has selected the corresponding DAC register at the Pixel Address write operation port 3C8H. The current read or write state of the DAC can be determined by examining the DAC State register.

Once the DAC is in a specific mode (read or write), an application can continue accessing the color registers by performing a sequence of three operations, one for each RGB value. The read sequence consists of selecting the desired DAC register in the Pixel Address register at the read operations port (3C7H) and then performing three consecutive IN instructions. The first one loads the 6-bit value stored in the DAC register for the color red, the second one loads the green value, and the third one load the blue value. The write sequence takes place in a similar fashion. This mode of operation allows rapid access to the three data items stored in each DAC register as well as to consecutive DAC registers. Because each entry in the DAC registers is six bits wide, the write operation is performed using the least significant six bits of each byte. The order of operations for the WRITE function is as follows:

1. Select the starting DAC color register number by means of a write operation to the Pixel Address write mode register at port 3C8H.
2. Disable interrupts.

3. Write the 18-bit color code in RGB encoding. The write sequence consists of three bytes consecutively output to the Pixel Data register. Only the six low-order bits in each byte are meaningful.
4. The DAC transfers the contents of the Pixel Data register to the DAC register number stored at the Pixel Address register.
5. The Pixel Address register increments automatically to point to the subsequent DAC register. Therefore, if more than one color is to be changed, the sequence of operations can be repeated from step 3.
6. Re-enable interrupts.

Read or write operations to the video DAC must be spaced 240 nanoseconds apart. Assembly language code can meet this timing requirement by inserting a short JMP instruction between successive IN or OUT opcodes. The instruction can be conveniently coded in this manner:

```
JMP        SHORT $ + 2; I/O delay
```

Chapter 6 lists examples of programming the DAC color registers.

Chapter

6

# VGA Drivers for Standard Modes

## 6.0 VGA Device Drivers

The animation programmer has a primary concern in the performance of the device driver routines. We have repeatedly mentioned that speed of execution is the factor that most often makes or breaks an animated application. For this reason, in animation the programmer must often squeeze the last microsecond of execution time from the CPU and the video hardware. In a VGA system this means selecting the most favorable display mode, designing and developing the most efficient device drivers, and resorting to every programming trick and stratagem that increases performance or improves the appearance of the animated display.

### 6.0.1 Standard Mode VGA Device Drivers

The VGA system can be considered as a different device in each operational mode. In fact, many VGA modes exist for no other reason than to provide compatibility with other devices. For this reason, the device drivers for VGA mode 18, with 640-by-480 pixels in 16 colors, are unrelated and incompatible with those for VGA mode 19, with 320-by-200 pixels in 256 colors, and both are different than the ones required for VGA nonstandard mode X. Since it is these three modes (modes 18, 19, and X) that provide the most powerful graphics functions in the VGA standard, and considering that compatibility with previous adapters is no longer a major consideration, the VGA drivers developed for this book are exclusively for the above-mentioned VGA modes 18, 19, and X.

In this chapter we develop the device drivers for VGA standard modes 18 and 19. Because graphics and animation programming in VGA mode X follow a slightly different methodology than in the standard modes, we have devoted Chapter 7 to mode X device drivers and fundamental routines.

## 6.1 VGA Programming Levels

Any display programming operations on a PC equipped with a VGA video system must inevitably access the VGA hardware or its memory space. Nevertheless, at the higher programming levels many of the details are hidden by the interface software. For example, a PC programmer working in Microsoft QuickBASIC has available a collection of program functions that allow drawing lines, boxes, circles, and ellipses, changing palette colors, performing fill operations, and even executing some primitive animation functions. The QuickBASIC programmer can perform all of the above mentioned graphics functions while ignoring the complications of VGA registers, video memory mapping, and DAC output. However, this transparency of hardware functions is achieved at a considerable cost in performance and control, a price that the animation programmer is not always willing or able to pay.

The programming levels in a PC equipped with VGA video are as follows:

1. VGA services provided by the operating system. This includes the video services in BIOS, MS-DOS, OS/2, WINDOWS, or other operating system programs or graphical environments.
2. VGA services provided by high-level languages and by programming libraries that extend the functions of high-level languages.
3. General-purpose VGA libraries that can be used directly or interfaced with one or more high-level languages.
4. Low-level routines, usually coded in assembly language, that access the VGA or DAC registers or the memory space reserved for video functions.

Observe that this list refers exclusively to the VGA system. Other graphics standards, such as 8514A, XGA, and SuperVGA, include high-level functions that are furnished as a programming interface with the hardware. The VGA standard does not furnish higher level programming facilities. In this chapter we discuss the lowest level of VGA programming, principally at the adapter hardware level. These lowest level services are often called *device driver* routines. VGA services in the BIOS are also used whenever these services do not compromise performance or control.

## 6.2 Device Drivers and Primitives

The term device driver is often used to denote the most elementary software elements that serve to isolate the operating system, or the high- and low-level programs, from the peculiarities of hardware devices and peripherals. It was the UNIX operating system that introduced the concept of an installable device driver. In UNIX a device driver is described as a software element that can be attached to the UNIX kernel at any time. The concept of a device driver was perpetuated by MS-DOS (starting with version 2.0) and by OS/2.

A second level of graphics routines, usually more elaborate than the device drivers, are called the graphics primitives. For example, to draw a circular arc on the graphics screen of a VGA system, we need to perform programming operations at two different levels. The higher level operation consists of calculating the $x$ and $y$ coordinates of the points that lay along the arc to be drawn. The second, and more elementary operation is to set to a desired color the screen pixels that lay along this arc. In this case we can say that the coordinate calculations for the arc are performed in a routine called a graphics primitive, while the setting of the individual screen pixels is left to a VGA device driver. In theory it is possible to reduce the device driver for a VGA graphic system to two routines: one to set to a predetermined color the screen pixel located at certain coordinates, and another one to read the color of a screen pixel. With the support of this simple, two-function driver, it is possible to develop VGA primitives to perform all the graphic functions of which the device is capable. Nevertheless, a system based on minimal drivers performs very poorly. For instance, a routine to fill a screen area with a certain color would have to make as many calls to the driver as there are pixels in the area to be filled. In practice, it is better to develop device drivers that perform more than minimum functions. Therefore, in addition to the pixel read and write services, it is convenient to include in the device driver category other elementary routines such as those that initialize the device and the video and operational modes, perform address calculations, read and write data in multipixel units, and manipulate the color settings at the system level.

### 6.2.1 VGA Device Drivers

In the PC under MS-DOS, the VGA graphics hardware is accessed by device drivers that are not installed as part of the operating system. Several interface mechanisms are possible for these drivers. One option is to link the graphics device driver to a software interrupt. Once this driver is loaded and its vector initialized, applications can access its services by means of the INT instruction. But this type of operation, while very convenient and efficient, requires that the driver be installed as a *terminate-and-stay-resident* (TSR) program, therefore reducing the memory available to applications. An alternative way of making the services of graphics device drivers accessible to applications is to include the drivers in graphics libraries. The library routines requested in the code, which are accessed by high- and low-level programs at link time, are incorporated into the program's run file. The procedures listed in this book can be incorporated into a library of this type.

#### Preparatory Operations

The code must perform certain preliminary operations before accessing the VGA graphics functions. Since graphics routines read and write the video buffer directly, it is usually convenient to set a segment register to the base address of video memory. In 8086 and 80286 systems the segment register most often available for this purpose is ES; in 80386, 486, and Pentium it is also possible to use FS or GS. The following procedure sets the ES register to the base address of the video buffer in a VGA graphics mode:

```
ES_TO_VIDEO     PROC    FAR
; Set the ES register to the base address of the video buffer
; in VGA graphics mode (A000H)
        PUSH    AX                  ; Save accumulator
        MOV     AX,0A000H           ; Video segment base for
                                    ; graphics modes
        MOV     ES,AX               ; To ES
        POP     AX                  ; Restore accumulator
        RET
ES_TO_VIDEO     ENDP
```

Another preparatory operation consists of setting the VGA video mode. If the selected video mode is a standard one, the easiest and most convenient way of setting it is by calling the corresponding BIOS service. Since this operation is usually not in a critical path of program performance, there is no substantial processing penalty from using the BIOS service. Nonstandard modes, on the other hand, usually require some customized code not provided in the BIOS. In Chapter 7 we list a routine to set VGA mode X using both BIOS and customized code. The following procedure can be used to set a VGA standard mode:

```
SET_MODE        PROC     FAR
; Set video display mode using BIOS service number 0 of INT 10H
; On entry:
;       AL = number of vide mode to set
; On exit:
;       carry clear
; Note: code assumes that the input is a valid mode number
;
        PUSH     AX                 ; Save AX
        MOV      AH,0               ; BIOS service request number
        INT      10H
        POP      AX                 ; Restore caller's register
        RET
SET_MODE        ENDP
```

Processing software often needs to know the current display mode so that it can be reset at the conclusion of program execution. The following procedure obtains the current VGA standard video mode:

```
GET_MODE        PROC      FAR
; Obtain current video mode using BIOS service number 15, INT 10H
; On entry:
;       nothing
; On exit:
;       AH = character columns on screen (40 or 80)
;          AL = active video mode
;          BH = active video page
;          carry clear
        MOV     AH,15           ; BIOS service request number
        INT     10H
        RET
GET_MODE        ENDP
```

## 6.2.2 VGA Mode 18 Pixel Write Routines

In VGA mode 18 each screen pixel is mapped to four memory maps, each map encoding the colors red, green, and blue, as well as the intensity component (see Figure 5.3). To set a screen pixel in this mode, the code must access individual bits located in four color maps. In Figure 5.3 the displayed screen pixel corresponds to the first bit in each of the four maps. But, due to the fact that the 80x86 instruction set does not contain operations for accessing individual bits, read and write operations in 80x86 assembly language must take place at the byte level. Consequently, the VGA program has to resort to bit masking. Figure 6.1 shows bit-to-pixel mapping in VGA mode 18.

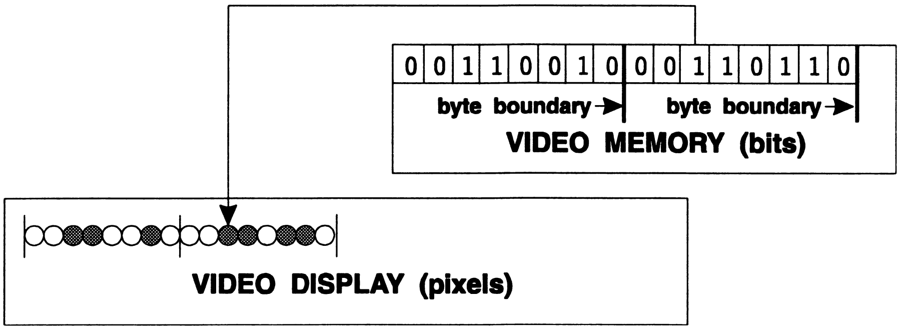


Figure 6.1 *Bit-to-Pixel Mapping in VGA Mode 18*

Notice in Figure 6.1 that the eleventh screen pixel (pointed at by the arrow) corresponds to the eleventh bit in the memory map. This eleventh bit is located in the second byte. We saw in Chapter 5 that VGA write operations can take place in four different write modes, labeled 0 to 3. Read operations can take place in either one of two modes, labeled 0 and 1. Code must determine and enable the desired read or write mode before accessing the VGA display functions. Therefore, it is necessary to develop a routine to set the desired read or write mode before pixel access can take place.

### Setting the Write Mode

To make the VGA more useful and flexible, its designers implemented several ways in which to write data to the video display. These are known as the *write* *modes*. VGA allows four different write modes, selected by means of bits 0 and 1 of the Graphics Mode register of the Graphics Controller (see Figure 5.10). The fundamental functions of the various write modes are as follows:

*Write mode 0* is the default mode. In write mode 0 the CPU, the Map Mask register of the Sequencer, and the Bit Mask register of the Graphics Controller are used to set a screen pixel to any desired color. Other VGA registers are also used for specific effects. For example, the Data Rotate register of the Graphics Controller has two fields which are significant during write mode 0 operations. The data rotate field (bits 0 to 3) determines how many positions to rotate the CPU data to the right before performing the write operation. The logical operation select field (bits 3 and 4) determines how the data stored in video memory is logically combined with the CPU data. The options are to write the CPU data unmodified or to AND, OR, or XOR it with the latched data. The VGA logical operations are discussed later in this section.

In *write mode 1* the contents of the latch registers, previously loaded by a read operation, are copied directly onto the color maps. Write mode 1, which is perhaps the simplest one, is often used in moving one area of video memory into another one. This write mode is particularly useful when the software takes advantage of the unused portions of video RAM. The location and amount of this unused memory varies in the different video modes. For example, in VGA graphics mode 18 the total pixel count is 38,400 pixels (640 pixels per row times 480 rows). Since the video buffer maps are 64K bytes, in each map there are 27,135 unused buffer bytes available to the programmer. This space can be used for storing images or data. On the other hand, video mode 19 consists of one byte per pixel and there are 320-by-200 screen pixels, totaling 64,000 bytes. Since the readily addressable area of the video buffer is limited to 65,536 bytes, the programmer has available only 1536 bytes for image manipulations.

*Write mode 2* is a simplified version of write mode 0. Like mode 0, it allows setting an individual pixel to any desired color. However, in write mode 2 the data rotate function (Data Rotate register) and the set-reset function (Set/Reset register) are not available. One advantage of write mode 2 over write mode 0 is its higher execution speed. Another difference between these write modes is that in write mode 2 the pixel color is determined by the contents of the CPU, and not by the setting of the Map Mask register or the Enable Set/Reset and Set/Reset registers. This characteristic simplifies coding and is one of the factors that determines the better performance of write mode 2.

In *write mode 3* the Data Rotate register of the Graphics Controller operates in the same manner as in write mode 0. The CPU data is ANDed with the Bit Mask register. The resulting bit pattern performs the same function as the Bit Mask register in write modes 0 and 2. The Set/Reset register also performs the same function as in write mode 0. However, the Enable Set/Reset register is not used. Therefore, the pixel color can be determined by programming either the Set/Reset register or the Map Mask register. The Map Mask register can also be programmed to selectively enable or disable the individual maps.

An application can use several read and write modes without fear of interference or conflict, since a change in the read or write mode does not affect the displayed image. On the other hand, a change in the video mode normally clears the screen and resets all VGA registers.

The write mode is selected by means of bits 0 and 1 of the Select Graphics Mode register of the Graphics Controller group (see Figure 5.10). Considering that the VGA behaves as a different device in each write mode, the device driver for a pixel write operation in mode 18 is write-mode specific. In other words, a different device driver is required for each write mode.

Each VGA write mode has its strong points, but it is generally accepted that write mode 2 is the most direct and generally useful one. In this write mode the individual pixel within a video buffer byte is selected by entering an appropriate mask in the Bit Mask register of the Graphics Controller. This bit mask contains a 1-bit for the pixel or pixels to be accessed and a 0 bit for those to be ignored. For example, the bit mask 00100000B can be used to select the pixel shown in Figure 6.1. The following procedure can be used to set a write mode while in display mode 18:

```
WRITE_MODE_18   PROC     FAR
; Set the Graphics Controller's Graphic Mode register to the
; desired write mode while in VGA mode 18
; On entry:
;         AL = write mode requested
; Note: Write mode bits are 0 and 1, all others are preserved
;
; Also set default bit mask
;
        MOV     AH,AL        ; Mode to AH
        MOV     DX,3CEH      ; Graphic Controller Address
                             ; register
        MOV     AL,5         ; Offset of the Mode Register
        OUT     DX,AL        ; Select this register
        INC     DX           ; Point to data register
; Read contents of Mode register
; Write modes bits are 000000xxB
        IN      AL,DX        ; AL has current bitmap
        AND     AL,11111100B ; Clear write mode bits
        OR      AL,AH        ; OR with write mode bits
                             ; passed by caller
; AL now holds bit pattern for Mode register
        OUT     DX,AL        ; Set register
; Set Bit Mask register to default setting
        MOV     DX,3CEH      ; Graphic Controller latch
        MOV     AL,8
        OUT     DX,AL        ; Select data register 8
        INC     DX           ; To 3CFH
        MOV     AL,0FFH      ; Default mask
        OUT     DX,AL        ; Load bit mask
        RET
WRITE_MODE_18   ENDP
```

The processing for setting a write mode while in VGA mode 19 is slightly different, since the code must also set bit 6 of the Select Graphics Mode register of the Graphics Controller group (see Figure 5.10). The following procedure sets the write mode in VGA mode 19:

```
WRITE_MODE_19     PROC      FAR
; Set the Graphics Controller's Graphic Mode register to the
; desired write mode in 256 colors
; On entry:
;         AL = write mode requested
; Also set default bit mask
;
        PUSH    AX              ; Save mode
        MOV     DX,3CEH         ; Graphic Controller Address
                                ; register
        MOV     AL,5            ; Offset of the Mode register
        OUT     DX,AL           ; Select this register
        INC     DX              ; Point to Data register (3CFH)
        POP     AX              ; Recover mode in AL
; Set bit 6 to enable 256 colors
        OR      AL,01000000B; Mask for bit 6
        OUT     DX,AL           ; Selected
; Set Bit Mask Register to default setting
        MOV     DX,3CEH         ; Graphic Controller latch
        MOV     AL,8
        OUT     DX,AL           ; Select Data register 8
        INC     DX              ; To 3CFH
        MOV     AL,0FFH         ; Default mask
        OUT     DX,AL           ; Load bit mask
        RET
WRITE_MODE_19   ENDP
```

## Setting the Read Mode

The VGA standard provides two different read modes. Read mode 0, which is the default, loads the CPU with the contents of one of the bit maps. In mode 18 we conventionally designate the color maps with the letters I, R, G, and B to represent the intensity, red, green, and blue elements. Which of the four maps is read into the CPU depends on the current setting of bits 0 and 1 of the Read Map Select register of the Graphics Controller (see Figure 5.9). In this context we sometimes say that the selected read map is *latched* onto the CPU. To read the contents of all four maps the program must execute four read operations to the same video buffer address; this is usually preceded by code to set the Read Map Select register.

Read mode 0 is useful in obtaining the contents of one or more video maps, while read mode 1 is more convenient when the programmer wishes to test for the presence of pixels that are set to a specific color or color pattern. In read mode 1 the contents of all four maps are compared with a predetermined mask.

This mask must have been stored beforehand in the Color Compare register of the Graphics Controller (see Figure 5.9). For example, to test for the presence of bright blue pixels, the IRGB bit pattern 1001B is stored in the Color Compare register. Thereafter, a single read operation appears to execute four successive logical ANDs with this mask. If a bit in any of the four maps matches the bit mask in the Color Compare register, it is set in the CPU; otherwise it is cleared.

The VGA read mode setting routine can be used with any standard video mode. The following procedure sets the desired read mode:

```
SET_READ_MODE   PROC    FAR
; Set the Graphics Controller Graphic Mode Select register to
; read mode 0 or 1. All other bits are preserved
; On entry:
;       AL = read mode requested
;
; Set bit 3 according to mode
;       Read mode 0 - bit 3 = 0 (AL = 00H)
;       Read mode 1 - bit 3 = 1 (AL = 08H)
;*********************|
;  test entry value   |
;*********************|
        CMP     AL,1            ; If entry value is not 1
        JNE     OK_BIT3         ; read mode 0 is forced
        MOV     AL,08H          ; 00001000B to set bit 3
;*********************|
;   set GC register   |
;*********************|
OK_BIT3:
        MOV     AH,AL           ; Bit mask to AH
        MOV     DX,3CEH         ; Graphic Controller address
                                ; register
        MOV     AL,5            ; Offset of the Mode register
        OUT     DX,AL           ; Select this register
        INC     DX              ; Point to Data register
; Read contents of mode register
        IN      AL,DX           ; AL has current bitmap
        AND     AL,11110111B    ; Clear bit 3
        OR      AL,AH           ; OR setting with mask
; AL now holds bit pattern for Mode register
        OUT     DX,AL           ; Set register
        RET
SET_READ_MODE       ENDP
```

## VGA Logical Operations

In Chapter 5 we saw that the Data Rotate register of the Graphics Controller determines how data is combined with data latched in the system microprocessor registers. The programmer can select the UNMODIFIED, AND, OR, and XOR logical operations by changing the value of bits 3 and 4 (see Figure 5.9).

Although all four logical operation modes find occasional use in VGA graphics programming, the XOR mode is particularly interesting. In animation routines the XOR mode provides a convenient way of drawing and erasing a screen object. The advantages of the XOR method are simpler and faster execution, and an easier way for restoring the original screen image. This is a convenient programming technique when more than one moving object can coincide on the same screen position.

One disadvantage of the XOR method is that the object's color depends on the color of the background over which it is displayed. If a graphics object is moved over different backgrounds, its color changes. Notice that some BIOS services set the Data Rotate register of the Graphics Controller to the normal mode. For example, if BIOS service number 9 of interrupt 10H is used to display text messages in a graphics application, the logical mode is automatically set to normal operation. Therefore, a program that uses the XOR, AND, or OR logical modes must reset the Data Rotate register after using this BIOS service. The subject of XOR animation is discussed at length in later chapters. The following procedures allow setting the VGA logical operations mode:

```
LOGICAL_OP      PROC      FAR
; Set the Graphics Controller Data Rotate register to the
; desired logical operation
; On entry:
;         AL = 0 to set data to UNMODIFIED operation
;         AL = 1 to AND with latched data
;         AL = 2 to OR with latched data
;         AL = 3 to XOR with latched data
;
        PUSH    CX                  ; Save caller's context
        MOV     CL,3                ; Number of bits to shift left
        SHL     AL,CL               ; Shift input 3 bit positions
                                    ; to reach bits 3 and 4
        PUSH    AX                  ; Save AL in stack
        MOV     DX,03CEH            ; Graphic Controller port address
        MOV     AL,3                ; Select Data Rotate register
        OUT     DX,AL
        INC     DX                  ; To data register
        POP     AX                  ; Restore AL
        OUT     DX,AL
        POP     CX                  ; Restore caller's context
        RET
LOGICAL_OP      ENDP
```

## Pixel Address Calculations

In Figure 6.1 the code must consider that the eleventh pixel is located in the second buffer byte. In VGA mode 18 this is usually accomplished by using either a word-size variable or an 80x86 machine register as an offset pointer. Since the VGA video buffer in a graphics mode starts at physical address A0000H, the ES register can be set to the corresponding segment base. The code to set the ES:BX register pair as a pointer to the second screen byte is as follows:

```
; Code fragment to set the 11th screen pixel while in VGA mode
; 18, write mode 2
         MOV     AX,0A000H       ; Segment base for video buffer
         MOV     ES,AX           ; To ES register
; ES — base of VGA video buffer
         MOV     BX,1            ; Offset of byte 2 to BX
; At this point ES:BX can be used to access the second byte in
; the video buffer
          .
          .
          .
```

A VGA mode 18 device driver should include a routine to calculate, from the pixel's screen coordinates, its offset and bit mask. The processing is based on the geometry of the video buffer in this mode, which is 80 bytes per screen row (640 pixels) and a total of 480 rows. The following procedure performs the necessary calculations:

```
PIXEL_ADD_18    PROC     FAR
; Address computation from x and y pixel coordinates
; On entry:
;               CX = x coordinate of pixel (range 0 to 639)
;               DX = y coordinate of pixel (range 0 to 479)
; On exit:
;               BX = byte offset into video buffer
;               AH = bit mask for the write operation using
;                    VGA write modes 0 or 2
;               AL is preserved
; Save all entry registers
        PUSH    CX
        PUSH    DX
;**********************|
;   calculate address  |
;**********************|
        PUSH    AX              ; Save accumulator
        PUSH    CX              ; Save x coordinate
        MOV     AX,DX           ; y coordinate to AX
        MOV     CX,80           ; Multiplier (80 bytes per row)
        MUL     CX              ; AX = y times 80
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
; Prepare for division
        MOV     CL,8            ; Divisor
        DIV     CL              ; AX / CL = quotient in AL and
                                ; remainder in AH
; Add in quotient
        MOV     CL,AH           ; Save remainder in CL
        MOV     AH,0            ; Clear high byte
        ADD     BX,AX           ; Offset into buffer to BX
        POP     AX              ; Restore AX
;**********************|
;   calculate bit mask  |
;**********************|
        MOV     AH,10000000B        ; Unit mask for 0 remainder
        SHR     AH,CL               ; Shift right CL times
; Restore all entry registers
        POP     DX
        POP     CX
        RET
PIXEL_ADD_18    ENDP
```

## Setting the Pixel

Once the bit mask and byte offset into the buffer have been determined, the code can then proceed to set an individual screen pixel. In VGA mode 18, write mode 2, this is accomplished in two steps: first the program sets the mask in the Bit Mask register of the Graphics Controller group, and then it performs a memory write operation to the address in ES:BX. The following procedure performs this function:

```
WRITE_PIX_18    PROC     FAR
; VGA mode 18 device driver for writing an individual
; pixel or a pixel pattern to the graphics screen
;
; On entry:
;           ES:BX = byte offset into the video buffer
;              AH = bit pattern to set
;                   (see PIXEL_ADD_18 procedure)
;              AL = pixel color in IRGB format
;
; This procedure assumes that write mode 2 is set
;
; Note: programs using this procedure usually precede its
;       call by one to ES_TO_VIDEO (to set the segment base)
;       and another one to PIXEL_ADD_18 (to obtain the byte
;       offset and pixel mask).
;       This procedure does not reset the default write mode nor
;       the contents of the Bit Mask register
;
        PUSH    DX              ; Save outer loop counter
        PUSH    AX              ; Color byte
        PUSH    AX              ; Twice
;**********************|
;     first step:       |
;    set bit mask       |
;**********************|
        MOV     DX,3CEH      ; Graphic controller latch
        MOV     AL,8
        OUT     DX,AL        ; Select data register 8
        INC     DX           ; To 3CFH
        POP     AX           ; AX once from stack
        MOV     AL,AH        ; Bit pattern
        OUT     DX,AL        ; Load bit mask
;
;**********************|
;       second step:   |
;    write IRGB color  |
;**********************|
        MOV     AL,ES:[BX]   ; Dummy read to load latch
                             ; registers
        POP     AX           ; Restore color code
        MOV     ES:[BX],AL   ; Write the pixel with the
                             ; color code in AL
        POP     DX           ; Restore outer loop counter
        RET
WRITE_PIX_18    ENDP
```

## Screen Tile Address Calculations

The finest possible degree of control over the VGA video display system is at the screen pixel level. However, it is often convenient to access the video display in units of several pixels, called pixel tiles or blocks. For example, when VGA mode 18 text display operations are performed by means of the BIOS character display services, these take place on a screen divided into 80 character columns and 30 character rows. This means that each character column is 8 pixels wide (640/80 = 8) and each row is 16 pixels high (480/30 = 16). In addition, graphics software can often benefit from operations that take place at coarser-than-pixel levels. For instance, in VGA mode 18, to draw a horizontal line from screen border to screen border requires 640 bit-level operations, but only 80 byte-level operations. Consequently, routines that read or write pixels in groups achieve substantially better performance than those that read or write the pixels individually. Animation programming can often take advantage of this increased performance.

In this book, for lack of a better word, we refer to 8-by-8 pixel units as *VGA screen tiles*, or simply tiles. Coarse-grain operations, in mode 18, see the video display as 80 columns and 60 rows of screen tiles. In this manner the programmer can envision the VGA screen in mode 18 either as consisting of 640-by-480 pixels (fine-grain visualization) or as consisting of 80-by-60 screen tiles of 8-by-8 pixels (coarse-grain visualization). Furthermore, the coarse grain visualization can easily be adapted to text display operations on an 80-by-30 screen by grouping the 60 tile rows into pairs. The following procedure calculates the coarse-grain offset into the video buffer from the vertical and horizontal tile count:

```
TILE_ADD_18     PROC    FAR
; Procedure to calculate the coarse-grain address at an 8-by-8
; pixel level in VGA mode 18
;
; On entry:
;       CH = horizontal tile number (range 0 to 79) = x coordinate
;       CL = vertical tile number (range 0 to 59) = y coordinate
;
; Compute coarse-grain address (in BX) as follows:
;       BX = (CL * 640) + CH
;
; On exit:
;       BX = tile offset into video buffer
;       CX is destroyed
;
        PUSH    AX          ; Save accumulator
        PUSH    DX          ; For word multiply
        PUSH    CX          ; To save CH for addition
        MOV     AX,CX       ; Copy CX in AX
; AL = CL
        MOV     AH,0        ; Clear high byte
        MOV     CX,640      ; CX is multiplier
        MUL     CX          ; AX * CX results in AX
; The multiplier (640) is the product of 80 tiles columns
; times 8 vertical pixels in each tile row
        POP     CX          ; Restore CH
        POP     DX          ; and DX
        MOV     CL,CH       ; Prepare to add in CH
        MOV     CH,0
        ADD     AX,CX       ; Add
        MOV     BX,AX       ; Move sum to BX
        POP     AX          ; Restore accumulator
        RET
TILE_ADD_18     ENDP
```

## Setting the Tile

Once the tile address has been determined, the individual tile (8-by-8 pixel group) can be set by placing an all-ones mask in the Bit Mask register of the Graphics Controller group, and then performing write operations to eight successive pixel rows. The following procedure sets the screen tile:

```
WRITE_TILE_18   PROC    FAR
; Write an 8-by-8 pixel block addressed at a coarse-grain level
; On entry:
;           ES:BX = byte offset into the video buffer
;                   (see TILE_ADD_18 procedure)
;              AL = pixel color in IRGB format
;
; This routine assumes that write mode 2 has been set
; Note: programs using this procedure usually precede its
;       call by one to ES_TO_VIDEO (to set the segment base)
;       and another one to TILE_ADD_18 (to obtain the
;       coarse-grain offset into the video buffer)
;       This procedure does not reset the default write mode nor
;       the contents of the Bit Mask register
;
        PUSH    DX              ; Save caller's context
        PUSH    CX
        PUSH    BX
        PUSH    AX              ; Color code byte
; Set Bit Mask register to all one bits
        MOV     DX,3CEH         ; Graphic Controller latch
        MOV     AL,8
        OUT     DX,AL           ; Select data register 8
        INC     DX              ; To 3CFH
        MOV     AL,0FFH         ; Bit pattern of all ones
        OUT     DX,AL           ; Load bit mask
; Set counter for 8 pixel rows
        MOV     CX,8            ; Counter initialized
        POP     AX              ; Restore color code
;*********************|
;     set 8 pixels    |
;*********************|
SET_EIGHT:
        MOV     AH,ES:[BX]      ; Dummy read to load latch
                                ; registers
        MOV     ES:[BX],AL      ; Write the pixel with the
                                ; color code in AL
        ADD     BX,80           ; Index to next row
        LOOP    SET_EIGHT
; Tile is set
        POP     BX              ; Restore caller's context
        POP     CX
        POP     DX
        RET
WRITE_TILE_18   ENDP
```

### 6.2.3 VGA Mode 18 Pixel Read Routine

A program attempting to determine the state of the eleventh pixel in Figure 6.1 would read the second memory byte and mask out all other bits. The mask, in this case, would have the value 00100000B. We have seen that video memory in VGA mode 18 is divided into four memory maps, labeled I, R, G, and B for the intensity, red, green, and blue components,respectively, also that all four maps are located at the same address. For this reason, in order to read the color code for an individual pixel, the program must successively select each of the four memory maps. This is done through the Read Operation Map Select register of the Graphics Controller, mentioned in Chapter 5. In other words, to determine the color of a single pixel in VGA mode 18, it is necessary to perform four separate read operations, one for each of the IRGB maps.

As in the write operation, the code to read a screen pixel must calculate the address of the video buffer byte in which the bit is located and the bit mask for isolating it. This can be done by means of the PIXEL_ADD_18 device driver previously listed. The following procedure reads a screen pixel in VGA mode 18 and returns the IRGB color value in the CL register:

```
READ_PIX_18      PROC    FAR
; Procedure to read a pixel color value in VGA mode 18
; On entry:
;            ES:BX = byte offset into the video buffer
;               AH = bit pattern for mask
; On exit:
;               CL = 4 low bits hold pixel color in IRGB format
;               CH = 0
; Code assumes that read mode 0 has been set
;
; Move bit mask to CH
        MOV     CH,AH         ; CH = bit mask for pixel
;*********************|
;   set up Sequencer   |
;*********************|
; Bit 3 of the Sequencer Memory Mode register must be clear for
; read mode 0 operations
        MOV     DX,3C4H       ; Sequencer Address register
        MOV     AL,4          ; Select Memory Mode register
        OUT     DX,AL         ; Activate
        INC     DX            ; To Sequencer Data register base
        IN      AL,DX         ; Read present value
        AND     AL,11110111B      ; Clear bit 3, preserve others
        OUT     DX,AL         ; Write to Memory Mode register
;**********************|
;   set up read loop    |
;**********************|
        MOV     AH,4          ; Reset Map Counter register
        MOV     CL,0          ; Clear pixel color return register
;**********************|
; execute 4 read cycles |
;**********************|
; AH has number for current IRGB map (range 0 to 3)
READ_MAPS:
; Select map from which to read
        MOV     DX,3CEH       ; Graphics Controller address
                              ; register
        MOV     AL,4          ; Read Map Select register
        OUT     DX,AL         ; Activate
        INC     DX            ; Graphics controller = 3CFH
        MOV     AL,AH         ; AH = counter for 4 maps
        DEC     AL          ; Adjust to range 0 to 3
        OUT     DX,AL       ; IRGB color map selected
;**********************|
;      read one byte    |
;**********************|
; Read 8 bits from selected map
        MOV     AL,ES:[BX]  ; Get byte from bit plane
;**********************|
; shift return register |
;**********************|
; Previous color code is in bit 0. The shift operation frees
; the low-order bit and moves previous bit codes to higher
; positions
        SHL     CL,1
;
;*********************|
;    mask out pixels   |
;*********************|
        AND     AL,CH       ; Pixel mask in CH
        JZ      NO_PIX_SET  ; Jump if no pixel in map
; Pixel was set in bitmap
        OR      CL,00000001B    ; Set bit 0 in Pixel Color
                                ; Return register
NO_PIX_SET:
        DEC     AH          ; Bump counter to next map
        JNZ     READ_MAPS   ; Continue if not last map
; 4 low bits in CL hold pixel color in IRGB format
        MOV     CH,0        ; Clear CH
        RET
READ_PIX_18     ENDP
```

### 6.2.4 VGA Mode 19 Pixel Write Routines

VGA programmers use mode 19 when screen color range is more important than definition. In this mode the VGA video display consists of 200 pixel rows of 320 pixels each. Each pixel, which can be in one of 256 colors, is determined by one byte in the video buffer. This scheme can be seen in Figure 5.4.

The fact that each screen pixel in mode 19 is mapped to a video buffer byte simplifies programming by eliminating the need for a bit mask. The VGA video buffer in mode 19 consists of 64,000 bytes. This number is the total pixel count obtained by multiplying the number of pixels per row by the number of screen rows ($320 \times 200 = 64{,}000$). Although the 64,000 buffer bytes are distributed in the four bit planes, the VGA hardware makes it appear to the programmer as if they resided in a continuous memory area. Thus, the top-left screen pixel is mapped to the byte at physical address A0000H, the next pixel on the top screen row is mapped to buffer address A0001H, and so forth. This byte-to-pixel mapping scheme can be seen in Figure 6.2.

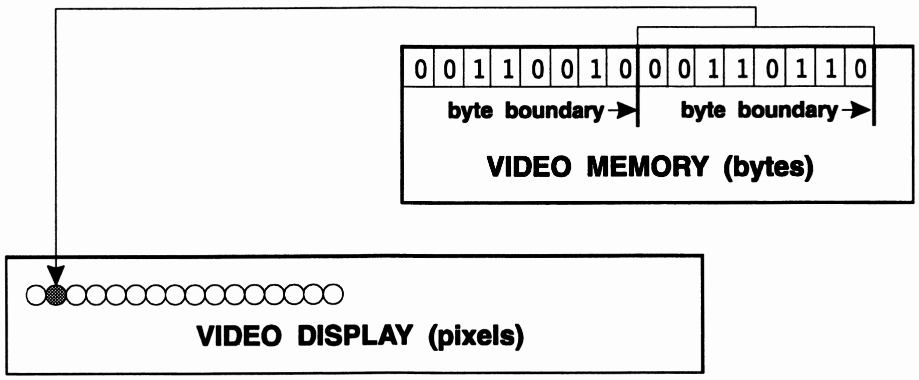


Figure 6.2 *Bit-to-Pixel Mapping in VGA Mode 19*

## Address Calculations

Address calculations in mode 19 are simpler than those in mode 18. All that is necessary to obtain the offset of a pixel into the video buffer is to multiply its row address by the number of buffer bytes per pixel row (320) and then add the pixel column. The processing is shown in the following procedure:

```
PIXEL_ADD_19    PROC    FAR
; Address computation for VGA mode 19
; On entry:
;               CX = x coordinate of pixel (range 0 to 319)
;               DX = y coordinate of pixel (range 0 to 199)
; On exit:
;               BX = offset into video buffer
;
        PUSH    AX              ; Save caller's context
        PUSH    CX
        PUSH    DX
        PUSH    CX              ; Save x coordinate
        MOV     AX,DX           ; y coordinate to AX
        MOV     CX,320          ; Multiplier is 320 bytes per row)
        MUL     CX              ; AX = y times 320
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
        ADD     BX,AX           ; Add in column value
        POP     DX              ; Restore caller's context
        POP     CX
        POP     AX
        RET
PIXEL_ADD_19    ENDP
```

### Setting the Pixel

Once the segment and the offset registers are loaded, the program can set an individual screen pixel by means of a simple MOV instruction. This means that the overhead of a pixel setting procedure is not necessary while programming in mode 19, since the code can set the desired pixel with a simple MOV instruction. The following code fragment shows the setting of a screen pixel:

```
; Write one pixel in VGA mode 19 (256 colors)
; Code assumes that write mode 0 for 256 colors is selected
;
; Register setup:
;       ES = A000H (video buffer segment base)
;       BX = offset into the video buffer (range 0 to 64000)
;       AL = 8-bit color code
;
        MOV    ES:[BX]AL       ; Write pixel
```

### 6.2.5 VGA Mode 19 Pixel Read Routine

In VGA mode 19 each screen pixel is mapped to a single video buffer byte. There are 64,000 bytes in the video buffer, which is the same as the total number of screen pixels obtained by multiplying the number of pixels per row by the number of screen rows (320 × 200 = 64,000). The mapping scheme in VGA mode 19 can be seen in Figure 6.2. The address calculations for mode 19 were shown in Section 6.2.3. The actual read operation is performed by means of a MOV instruction, as in the following code fragment:

```
; Read one pixel in VGA mode 19 (256 colors)
; Code assumes that read mode 0 is selected
;
; Register setup:
;      ES = A000H (video buffer segment base)
;      BX = offset into the video buffer (range 0 to 64000)
;
        MOV    AL,BYTE PTR ES:[BX]    ; Read pixel
; AL now holds the 8-bit color code
```

## 6.3 Color Operations

The theory of additive color reproduction is based on the fact that light in the primary colors (red, green, and blue) can be used to generate all the colors of the spectrum. Red, green, and blue are called the *primary colors*. Also notice that, technically, it is possible to create white light by blending just two colors. The color that must be blended with a primary color to form white is called the complement of the primary color, or the *complementary color*. Figure 6.3 is a diagram of the additive primary and complementary colors.

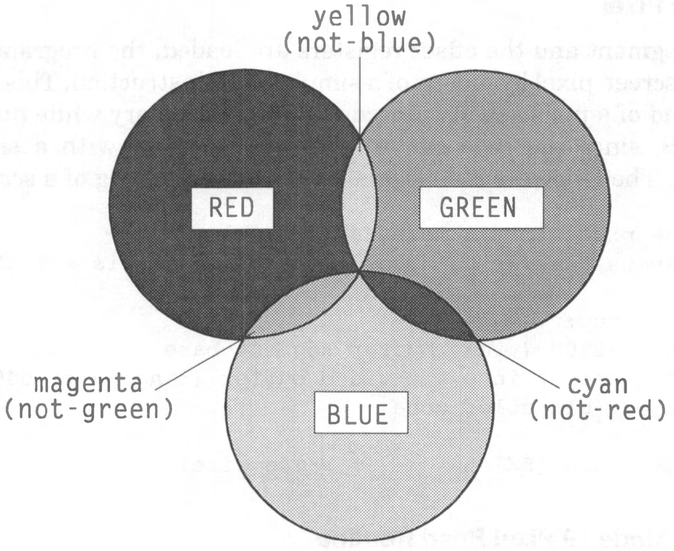


Figure 6.3 *Additive Primary and Complementary Colors*

The complementary colors can also be described as white light minus a primary color. For example, white light without red, not-red, gives a shade of blue-green known as cyan; not-green gives a mixture of red and blue called magenta; and not-blue gives yellow, which is a mixture of red and green light (see Figure 6.3). Video display technology is usually designed on additive color blending. Subtractive methods are based on dyes that absorb the undesirable, complementary colors. A cyan-colored filter, for example, absorbs the green and blue components of white light. Subtractive mixing is used in color photography and color printing.

In describing a color we use three characteristics that can be precisely determined: its *hue*, its *intensity*, and its *saturation*. A method of color measurement based on hue, intensity, and saturation (sometimes called the *HIS*) was developed for color television. The hue can be defined as the color of a color. Physically the hue can be measured by the color's dominant wavelength. The intensity of a color is its brightness. This brightness is measured in units of luminance or *nits*. The saturation of a color is its purity. If the color contains no white diluent, it is said to be fully saturated.

### 6.3.1 256-Color Modes

While address calculations in VGA mode 19 are simpler than in mode 18, the pixel color encoding is considerably more complicated. This is so not only because there is a more extensive color range in mode 19 than in mode 18 (256 versus 16 colors) but also because the default encoding scheme is not very straightforward. This default scheme is determined by the setting of the 256

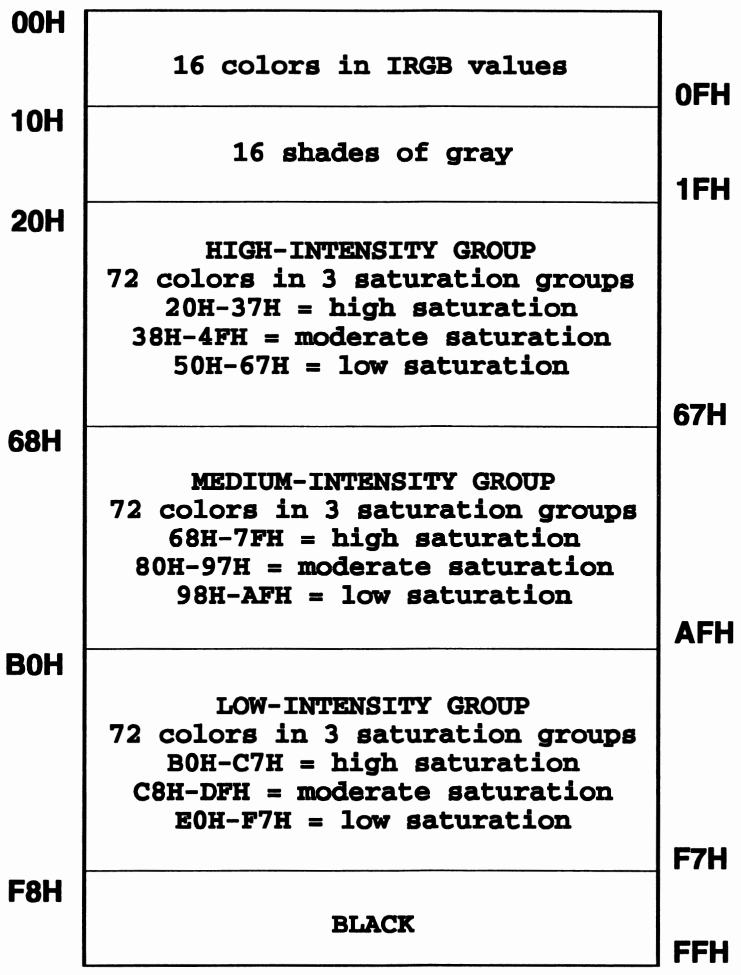

00H
16 colors in IRGB values
0FH
10H
16 shades of gray
1FH
20H
HIGH-INTENSITY GROUP
72 colors in 3 saturation groups
20H-37H = high saturation
38H-4FH = moderate saturation
50H-67H = low saturation
67H
68H
MEDIUM-INTENSITY GROUP
72 colors in 3 saturation groups
68H-7FH = high saturation
80H-97H = moderate saturation
98H-AFH = low saturation
AFH
B0H
LOW-INTENSITY GROUP
72 colors in 3 saturation groups
B0H-C7H = high saturation
C8H-DFH = moderate saturation
E0H-F7H = low saturation
F7H
F8H
BLACK
FFH

Figure 6.4 *Default Color Register Setting in VGA Mode 19*

Color registers in the DAC (see Section 5.3). The start-up value stored in these registers by the BIOS initialization code is designed to provide compatibility with the CGA and EGA systems. Figure 6.4 shows the default setting of the DAC Color registers in VGA mode 19.

In Figure 6.4 the first group of default colors (range 00H to 0FH) corresponds to those in the 16-color VGA modes. This design ensures that if only the four low-order bits of the 8-bit color code are programmed, the resulting colors in the 256-color mode are the same as those in the 16-color modes. The second group of default colors (range 10H to 1FH) corresponds to 16 shades of gray. The next group of colors (range 20H to 67H) consists of 72 colors divided into three subgroups, each one representing a different level of color saturation. Each of the saturation subgroups consists of 24 colors in a circular pattern of blue-red-green hues. Another 72-color group is used for medium-intensity colors and a third one for low-intensity colors.

The VGA programmer is by no means restricted to the default values installed by the BIOS in the DAC Color registers. This default setting is not convenient for many applications. The main objection to the default 256-color map is that red, green, and blue are not mapped to adjacent bits or located in manageable fields. For example, using the default setting of the DAC Color registers, the various shades of the color green are obtained with the values shown in Table 6.1.

Table 6.1 *Default Shades of Green in VGA 256-Color Mode*

| VALUE/RANGE | | INTENSITY | SATURATION |
|---|---|---|---|
| 02H | 00000010B | medium | high |
| 0AH | 00001010B | high | high |
| 2EH to 34H | 00101110B to 00110100B | high | high |
| 46H to 4CH | 01000110B to 01001100B | high | moderate |
| 5EH to 64H | 01011110B to 01100100B | high | low |
| 76H to 7CH | 01110110B to 01111100B | medium | high |
| 8EH to 94H | 10001110B to 10010100B | medium | moderate |
| A6H to ACH | 10100110B to 10101100B | medium | low |
| BEH to C4H | 10111110B to 11000100B | low | high |
| D6H to DCH | 11010110B to 11011100B | low | moderate |
| EEH to F4H | 11101110B to 11110100B | low | low |

If compatibility with previous video standards is not at issue, a more rational 256-color scheme can be based on assigning 2 bits to each of the components of the familiar IRGB encoding. Figure 6.5 shows the bit-mapping for this IRGB double-bit encoding.

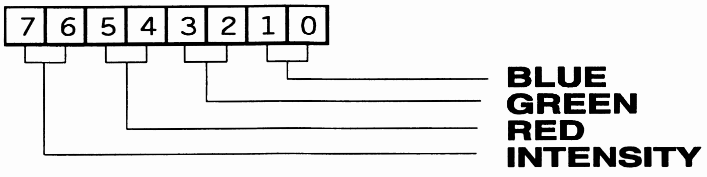


Figure 6.5 *Double-Bit Mapping for 256-Color Modes*

To enable the double-bit encoding in Figure 6.5, it is necessary to change the default values in the DAC registers. In Chapter 5 we saw that the DAC Color registers consist of 18 bits, 6 bits for each color (red, green, and blue). The bitmap of the DAC Color registers is shown in Figure 6.6.

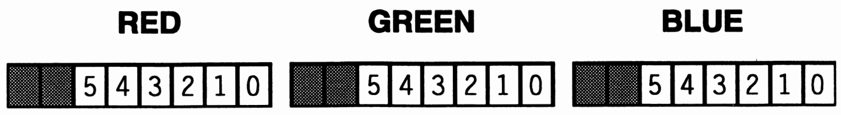


Figure 6.6 *DAC Color Register Bitmap*

To design an 8-bit encoding in a four-element (IRGB) format we have assigned 2 bits to each color and to the intensity component (see Figure 6.5). In this manner, the 2-bit values for red, green, and blue allow four tones. Since each tone can be in four brightness levels, one for each intensity bit setting, each pure hue would have 16 saturations. In order to achieve a double-bit IRGB encoding by reprogramming the DAC Color registers (see Figure 6.6), we assign eight values to each DAC Color register, as shown in Table 6.2.

Table 6.2 *DAC Register Setting for Double-Bit IRGB Encoding*

| NUMBER | 6-BIT VALUE | INTENSITY | COLOR |
|---|---|---|---|
| 0 | 9 | OFF | dark |
| 1 | 18 | OFF | . |
| 2 | 27 | OFF | . |
| 3 | 36 | OFF | . |
| 4 | 45 | ON | . |
| 5 | 54 | ON | . |
| 6 | 63 | ON | bright |

The first four bit settings in Table 6.2 correspond to the color tones controlled by the red, green, and blue bits when the intensity bits have a value of 00B. The last three 6-bit values correspond to the three additional levels of intensity. Excluding the intensity bit, the three DAC Color registers have 64 possible combinations. Table 6.3 shows the pattern of register settings for the double-bit IRGB format.

Table 6.3 *Pattern for DAC Register Settings in Double-Bit IRGB Encoding*

| I = 00 | | | | I = 01 | | | | I = 10 | | | | I = 11 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. | R | G | B | No. | R | G | B | No. | R | G | B | No. | R | G | B |
| 0 | 9 | 9 | 9 | 64 | 9 | 9 | 18 | 128 | 9 | 9 | 27 | 192 | 9 | 9 | 36 |
| 1 | 9 | 9 | 18 | 65 | 9 | 9 | 27 | 129 | 9 | 9 | 36 | 193 | 9 | 9 | 45 |
| 2 | 9 | 9 | 27 | 66 | 9 | 9 | 36 | 130 | 9 | 9 | 45 | 194 | 9 | 9 | 54 |
| 3 | 9 | 9 | 36 | 67 | 9 | 9 | 45 | 131 | 9 | 9 | 54 | 195 | 9 | 9 | 63 |
| 4 | 9 | 9 | 9 | 68 | 9 | 18 | 18 | 132 | 9 | 27 | 18 | 196 | 9 | 36 | 18 |
| 5 | 9 | 18 | 9 | 69 | 9 | 27 | 18 | 133 | 9 | 36 | 27 | 197 | 9 | 45 | 36 |
| . | | . | | | . | | | | . | | | | . | | |
| 63 | 36 | 36 | 36 | 127 | 45 | 45 | 45 | 191 | 54 | 54 | 54 | 255 | 63 | 63 | 63 |

In Table 6.3 a value of 9 in the red, green, and blue color registers corresponds to the color black. It has been found that the colors generated by the low range of the DAC scale are less noticeable than those on the high range. By equating the value 9 to the color black we enhance the visible color range on a standard VGA. The following procedure changes the default setting of the DAC Color registers to the values in Table 6.3:

```
TWO_BIT_IRGB    PROC    FAR
; Initialize DAC registers for 256-color mode in the following
; format:
;               7 6 5 4 3 2 1 0  <= bits
;               |_| |_| |_| |_|
;                I   R   G   B
;
; Note: data for this procedure follows the code listing
;
;*********************|
;    save caller's    |
;      context        |
;*********************|
        PUSH    ES                  ; Registers used by routine
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
        PUSH    BP
;*********************|
;    set ES to CS     |
;*********************|
        MOV     AX,P_DATA           ; Local data segment
        MOV     ES,AX               ; To DS
        ASSUME  ES:P_DATA           ; Assume this DS
;
;*********************|
;  expand color table |
;*********************|
; The code segment table named INTENSITY_0 contains the register
; values for 00 intensity bits. The values for 01, 10, and 11
; intensity bits are calculated by adding 9, 18, and 27 to the
; values in the INTENSITY_0 table
        LEA     SI,ES:INTENSITY_0 ; Set pointer to source table
        LEA     DI,ES:INTENSITY_1 ; Pointer to intensity 01 table
        LEA     BX,ES:INTENSITY_2 ; Pointer to intensity 10 table
        LEA     BP,ES:INTENSITY_3 ; Pointer to intensity 11 table
        MOV     CX,192           ; Number of values to move
        MOV     AH,9             ; Constant value to add
DO_TABLES:
        MOV     AL,ES:[SI]       ; Value from intensity 00 table
        ADD     AL,AH            ; Add 9, once
        MOV     ES:[DI],AL       ; Place in intensity 01 table
        ADD     AL,AH            ; Add 18
        MOV     ES:[BX],AL       ; Place in intensity 10 table
        ADD     AL,AH            ; Add 27
        MOV     ES:[BP],AL       ; Place in intensity 11 table
        INC     SI               ; Bump all pointers
        INC     DI
        INC     BX
        INC     BP
        LOOP    DO_TABLES
;*********************|
;  set DAC registers  |
;*********************|
; Change first 64 DAC registers using BIOS service number 16
; of interrupt 10H
        MOV     AH,16           ; Service number
        MOV     AL,18           ; Subservice to set group of
                                ; DAC color registers
        LEA     DX,ES:INTENSITY_0  ; Pointer to table of 256 DAC
                                ; colors
        MOV     BX,0            ; Start with register
        MOV     CX,256          ; Set all 256 DAC registers
        INT     10H
;*********************|
;   restore caller's  |
;       context       |
;*********************|
        POP     BP              ; Registers in stack
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        RET
TWO_BIT_IRGB    ENDP
;****************************************************************
;               data for TWO_BIT_IRGB procedure
;****************************************************************
P_DATA  SEGMENT
;                       | R   G   B | R   G   B | R   G   B |
INTENSITY_0     DB      009,009,009,009,009,018,009,009,027 ; 2
                DB      009,009,036,009,018,009,009,018,018 ; 5
                DB      009,018,027,009,018,036,009,027,009 ; 8
                DB      009,027,018,009,027,027,009,027,036 ; 11
                DB      009,036,009,009,036,018,009,036,027 ; 14
                DB      009,036,036,018,009,009,018,009,018 ; 17
                DB      018,009,027,018,009,036,018,018,009 ; 20
                DB      018,018,018,018,018,027,018,018,036 ; 23
                DB      018,027,009,018,027,018,018,027,027 ; 26
                DB      018,027,036,018,036,009,018,036,018 ; 29
                DB      018,036,027,018,036,036,027,009,009 ; 32
                DB      027,009,018,027,009,027,027,009,036 ; 35
                DB      027,018,009,027,018,018,027,018,027 ; 38
                DB      027,018,036,027,027,009,027,027,018 ; 41
                DB      027,027,027,027,027,036,009,036,009 ; 44
                DB      027,036,018,027,036,027,027,036,036 ; 47
                DB      036,009,009,036,009,018,036,009,027 ; 50
                DB      036,009,036,036,018,009,036,018,018 ; 53
                DB      036,018,027,036,018,036,036,027,009 ; 56
                DB      036,027,018,036,027,027,036,027,036 ; 59
                DB      036,036,009,036,036,018,036,036,027 ; 62
                DB      036,036,036;                        ; 63
;
INTENSITY_1     DB      192 DUP (00)
INTENSITY_2     DB      192 DUP (00)
INTENSITY_3     DB      192 DUP (00)
;
P_DATA   ENDS
```

A double-bit IRGB setting for the DAC registers simplifies programming in the VGA 256-color mode when compared to the default setting. Once the DAC registers are set for the double-bit IRGB encoding the programmer can choose any one color by setting the corresponding bits in the video buffer byte mapped to the pixel. For example, the bit combinations in Table 6.4 can be used to display 16 pure tones of magenta. Notice that the purity of the hue is ensured by the zero value in the green DAC register.

Table 6.4 *16 Shades of Magenta Using Double-bit IRGB Code*

| NUMBER | I | R | G | B | TONE |
|---|---|---|---|---|---|
| 0 | 00 | 01 | 00 | 01 | darkest magenta |
| 1 | 00 | 10 | 00 | 10 | . |
| 2 | 00 | 01 | 00 | 01 | . |
| 3 | 00 | 11 | 00 | 11 | . |
| 4 | 01 | 01 | 00 | 01 | . |
| . |  | . |  |  | . |
| . |  | . |  |  | . |
| 15 | 11 | 11 | 00 | 11 | brightest magenta |

No single color encoding is ideal for all purposes. Often the programmer prefers to enhance certain portions of the color range at the expense of other portions. For example, in displaying a mountain landscape it might be preferable to extend shades of blue and green at the expense of red. On the other hand, a volcanic explosion may require more shades of red than of green and blue. The programmer can manipulate the displayed range by choosing which set of 256 colors, from a possible total of 262,143, is installed in the DAC Color registers.

## Shades of Gray

Gray is defined as equal intensities of the primary colors, red, green, and blue. In the DAC Color registers any setting in which the three values are equal generates a shade of gray. For example, the value 20-20-20, for red-green-blue, respectively, produces a 31 percent gray shade, while a value of 32-32-32 produces a 50 percent gray shade. Recall that each register can hold 64 values. Since the gray shades require that all three colors have the same value, there are 64 possible shades of gray in the VGA 256-color modes. The actual settings of the VGA registers go from 0-0-0 to 63-63-63.

A graphics program operating in VGA 256-color mode can simultaneously use the full range of 64 gray shades, as well as 192 additional colors. This requires reprogramming the DAC Color registers. If a program were to execute in shades of gray only, then the low-order 6 bits of the color encoding can be used to select the gray shades. The range would extend from 0 (black) to a value of 63 (brightest white). The setting of the DAC Color registers for a 64-step grayscale is shown in Table 6.5.

Table 6.5 *Pattern for DAC Register Setting for 64 Shades of Gray*

| No. | R | G | B | No. | R | G | B | No. | R | G | B | No. | R | G | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 64 | 0 | 0 | 0 | 128 | 0 | 0 | 0 | 192 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 65 | 1 | 1 | 1 | 129 | 1 | 1 | 1 | 193 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 | 66 | 2 | 2 | 2 | 130 | 2 | 2 | 2 | 194 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 67 | 3 | 3 | 3 | 131 | 3 | 3 | 3 | 195 | 3 | 3 | 3 |
| ⋮ | ⋮ | | | | ⋮ | | | | ⋮ | | | | ⋮ | | |
| 63 | 63 | 63 | 63 | 127 | 63 | 63 | 63 | 191 | 54 | 54 | 54 | 255 | 63 | 63 | 63 |

Notice in Table 6.5 that the gray settings are repeated four times. The effect of this repeated pattern is that the high-order bits of the color code are ignored. In other words, all possible color values generate a gray shade, and the excess of 63 (00111111B) has no visible effect. The following procedure changes the default setting of the DAC Color registers to the values in Table 6.5:

```
GRAY_256        PROC    FAR
; Initialize DAC registers for 64 shades of gray, as follows:
;               7 6 5 4 3 2 1 0   <= bits
;               | | |_|_|_|_|_|___ Grayscale in the range 0-63
;               | |
;               |_|_______________ Not significant
;
; Notes: This procedure sets four blocks of 64 DAC registers to
;        the same gray scale so that bits 6 and 7 of video color
;        data are ignored
;        Data for this procedure follows the code listing
;
;
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    ES              ; Registers used by routine
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
;**********************|
;    set ES to CS      |
;**********************|
        MOV     AX,P_DATA        ; Local data segment
        MOV     ES,AX            ; To DS
        ASSUME  ES:P_DATA        ; Assume this ES
;**********************|
;  set DAC registers   |
;**********************|
; Change first 256 DAC registers using BIOS service number 16
; of interrupt 10H. DAC registers are set to a 64-step
; grayscale, repeated four times
        MOV     AH,16            ; Service number
        MOV     AL,18            ; Subservice to set group of
                                 ; DAC color registers
        LEA     DX,ES:GRAY_MAP   ; Pointer to table of 256 DAC
                                 ; colors, in groups of 64 shades
                                 ; of gray
        MOV     BX,0             ; Start with register
        MOV     CX,256           ; Set 256 DAC registers
        INT     10H
;**********************|
;   restore caller's   |
;       context        |
;**********************|
        POP     DX               ; Registers in stack
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        RET
GRAY_256        ENDP

;****************************************************************
;                data for the procedure GRAY_256
;****************************************************************
P_DATA  SEGMENT
;                      | R   G   B | R   G   B | R   G   B |
GRAY_MAP        DB      000,000,000,001,001,001,002,002,002 ; 2
                DB      003,003,003,004,004,004,005,005,005 ; 5
                DB      006,006,006,007,007,007,008,008,008 ; 8
                DB      009,009,009,010,010,010,011,011,011 ; 11
                DB      012,012,012,013,013,013,014,014,014 ; 14
                DB      015,015,015,016,016,016,017,017,017 ; 17
                DB      018,018,018,019,019,019,020,020,020 ; 20
                DB      021,021,021,022,022,022,023,023,023 ; 23
                DB      024,024,024,025,025,025,026,026,026 ; 26
                DB      027,027,027,028,028,028,029,029,029 ; 29
                DB      030,030,030,031,031,031,032,032,032 ; 32
                DB      033,033,033,034,034,034,035,035,035 ; 35
                DB      036,036,036,037,037,037,038,038,038 ; 38
                DB      039,039,039,040,040,040,041,041,041 ; 41
                DB      042,042,042,043,043,043,044,044,044 ; 44
                DB      045,045,045,046,046,046,047,047,047 ; 47
        DB      048,048,048,049,049,049,050,050,050 ; 50
        DB      051,051,051,052,052,052,053,053,053 ; 53
        DB      054,054,054,055,055,055,056,056,056 ; 56
        DB      057,057,057,058,058,058,059,059,059 ; 59
        DB      060,060,060,061,061,061,062,062,062 ; 62
        DB      063,063,063                         ; 63
        DB      000,000,000,001,001,001,002,002,002 ; 2
        DB      003,003,003,004,004,004,005,005,005 ; 5
        DB      006,006,006,007,007,007,008,008,008 ; 8
        DB      009,009,009,010,010,010,011,011,011 ; 11
        DB      012,012,012,013,013,013,014,014,014 ; 14
        DB      015,015,015,016,016,016,017,017,017 ; 17
        DB      018,018,018,019,019,019,020,020,020 ; 20
        DB      021,021,021,022,022,022,023,023,023 ; 23
        DB      024,024,024,025,025,025,026,026,026 ; 26
        DB      027,027,027,028,028,028,029,029,029 ; 29
        DB      030,030,030,031,031,031,032,032,032 ; 32
        DB      033,033,033,034,034,034,035,035,035 ; 35
        DB      036,036,036,037,037,037,038,038,038 ; 38
        DB      039,039,039,040,040,040,041,041,041 ; 41
        DB      042,042,042,043,043,043,044,044,044 ; 44
        DB      045,045,045,046,046,046,047,047,047 ; 47
        DB      048,048,048,049,049,049,050,050,050 ; 50
        DB      051,051,051,052,052,052,053,053,053 ; 53
        DB      054,054,054,055,055,055,056,056,056 ; 56
        DB      057,057,057,058,058,058,059,059,059 ; 59
        DB      060,060,060,061,061,061,062,062,062 ; 62
        DB      063,063,063                         ; 63
        DB      000,000,000,001,001,001,002,002,002 ; 2
        DB      003,003,003,004,004,004,005,005,005 ; 5
        DB      006,006,006,007,007,007,008,008,008 ; 8
        DB      009,009,009,010,010,010,011,011,011 ; 11
        DB      012,012,012,013,013,013,014,014,014 ; 14
        DB      015,015,015,016,016,016,017,017,017 ; 17
        DB      018,018,018,019,019,019,020,020,020 ; 20
        DB      021,021,021,022,022,022,023,023,023 ; 23
        DB      024,024,024,025,025,025,026,026,026 ; 26
        DB      027,027,027,028,028,028,029,029,029 ; 29
        DB      030,030,030,031,031,031,032,032,032 ; 32
        DB      033,033,033,034,034,034,035,035,035 ; 35
        DB      036,036,036,037,037,037,038,038,038 ; 38
        DB      039,039,039,040,040,040,041,041,041 ; 41
        DB      042,042,042,043,043,043,044,044,044 ; 44
        DB      045,045,045,046,046,046,047,047,047 ; 47
        DB      048,048,048,049,049,049,050,050,050 ; 50
        DB      051,051,051,052,052,052,053,053,053 ; 53
        DB      054,054,054,055,055,055,056,056,056 ; 56
        DB      057,057,057,058,058,058,059,059,059 ; 59
        DB      060,060,060,061,061,061,062,062,062 ; 62
        DB      063,063,063                         ; 63
        DB      000,000,000,001,001,001,002,002,002 ; 2
```

```
                  DB      003,003,003,004,004,004,005,005,005 ; 5
                  DB      006,006,006,007,007,007,008,008,008 ; 8
                  DB      009,009,009,010,010,010,011,011,011 ; 11
                  DB      012,012,012,013,013,013,014,014,014 ; 14
                  DB      015,015,015,016,016,016,017,017,017 ; 17
                  DB      018,018,018,019,019,019,020,020,020 ; 20
                  DB      021,021,021,022,022,022,023,023,023 ; 23
                  DB      024,024,024,025,025,025,026,026,026 ; 26
                  DB      027,027,027,028,028,028,029,029,029 ; 29
                  DB      030,030,030,031,031,031,032,032,032 ; 32
                  DB      033,033,033,034,034,034,035,035,035 ; 35
                  DB      036,036,036,037,037,037,038,038,038 ; 38
                  DB      039,039,039,040,040,040,041,041,041 ; 41
                  DB      042,042,042,043,043,043,044,044,044 ; 44
                  DB      045,045,045,046,046,046,047,047,047 ; 47
                  DB      048,048,048,049,049,049,050,050,050 ; 50
                  DB      051,051,051,052,052,052,053,053,053 ; 53
                  DB      054,054,054,055,055,055,056,056,056 ; 56
                  DB      057,057,057,058,058,058,059,059,059 ; 59
                  DB      060,060,060,061,061,061,062,062,062 ; 62
                  DB      063,063,063                         ; 63
;
P_DATA  ENDS
```

## Summing to Gray Shades

A program can read the red, green, and blue values installed in a DAC Color register and find an equivalent gray shade with which to replace it. If this action is performed simultaneously on all 256 DAC Color registers, the result converts a displayed color image to black-and-white. Considering that the human eye is more sensitive to certain regions of the spectrum, this conversion is usually based on assigning different weights to the red, green, and blue components. This relative color weight is used to determine the gray shade, on a scale of 0 to 63. As mentioned in the previous paragraph, the resulting grayscale setting must have equal proportions of the red, green, and blue elements.

BIOS service number 16, of interrupt 10H, contains subservice number 27, which sums all color values in the DAC registers to gray shades. The BIOS code uses a weighted sum based on the following values:

```
  red = 30%
green = 59%
 blue = 11%
------
total = 100%
```

One disadvantage in using the BIOS service is that it does not preserve the original values found in the DAC registers. The following procedure performs a gray scale sum based on the action of the above-mentioned BIOS service:

```
SUM_TO_GRAY     PROC    FAR
; Uses BIOS service number 16, subservice number 27, to perform
; a sum-to-gray-shades operation on all 256 DAC Color registers
; The entry values in the DAC registers are not preserved
;
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    AX                ; Save registers used by routine
        PUSH    BX
        PUSH    CX
; Initialize registers for BIOS service
        MOV     AH,16             ; Service request number
        MOV     AL,27             ; Subservice is sum-to-gray
        MOV     BX,0              ; Start with first register
        MOV     CX,256            ; All 256 DAC Color registers
        INT     10H
;**********************|
;   restore caller's   |
;       context        |
;**********************|
        POP     CX                ; Caller's registers in stack
        POP     BX
        POP     AX
        RET
SUM_TO_GRAY     ENDP
```

Preserving the DAC registers is a simple manipulation performed by BIOS service number 16, subservice 23, of INT 10H. The following procedures perform the DAC save operation:

```
SAVE_DAC                PROC    FAR
; Save current values in DAC Color registers
; Data is saved in the local segment named P_DATA, under the
; variable name DAC_REGS (see data area following code listing)
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    ES                ; Registers used by routine
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
;**********************|
;    set ES to CS      |
;**********************|
        MOV     AX,P_DATA         ; Local data segment
        MOV     ES,AX             ; To DS
        ASSUME  ES:P_DATA         ; Assume this DS
; Initialize registers for BIOS service
        MOV     AH,16             ; Service request number
        MOV     AL,23            ; Subservice is read DAC group
        MOV     BX,0             ; Start with first register
        MOV     CX,256           ; All 256 DAC Color registers
        LEA     DX,ES:DAC_REGS   ; Pointer to storage area
        INT     10H
;**********************|
;   restore caller's   |
;       context        |
;**********************|
        POP     DX               ; Registers in stack
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        RET
SAVE_DAC        ENDP
```

Restoring the DAC registers is performed by BIOS service number 16, subservice 18, of INT 10H. The following procedures perform the DAC restore operation:

```
RESTORE_DAC     PROC    FAR
; Restore DAC Color registers to values stored by the SAVE_DAC
; procedure
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    ES               ; Registers used by routine
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
;**********************|
;    set ES to CS      |
;**********************|
        MOV     AX,P_DATA        ; Local data segment
        MOV     ES,AX            ; To DS
        ASSUME  ES:P_DATA        ; Assume this DS
; Initialize registers for BIOS service
        MOV     AH,16            ; Service request number
        MOV     AL,18            ; Subservice is write DAC group
        MOV     BX,0             ; Start with first register
        MOV     CX,256           ; All 256 DAC Color registers
        LEA     DX,ES:DAC_REGS   ; Pointer to storage area
        INT     10H
;**********************|
;   restore caller's   |
;       context        |
;**********************|
        POP     DX              ; Registers in stack
        POP     CX
        POP     BX
        POP     AX
        POP     ES
        RET
RESTORE_DAC     ENDP

;****************************************************************
;        data for the procedures SAVE_DAC and RESTORE_DAC
;****************************************************************
P_DATA  SEGMENT
;
DAC_REGS        DB      768 DUP (00H)   ; Storage for 256
                                ; registers in 3 color bytes per
                                ; register
                DW      0       ; Padding
;
P_DATA  ENDS
```

The IBM BIOS performs several automatic operations on the VGA DAC Color registers. For example, during a mode change call (BIOS service number 0, interrupt 10H) the BIOS loads all 256 DAC Color registers with the default values. If the mode change is to a monochrome mode, then a sum-to-gray operation is performed. The programmer can prevent this automatic loading of the DAC registers. BIOS service number 18, subservice number 49, of interrupt 10H, enables and disables the default pallete loading during mode changes. Subservice number 51 enables and disables the sum-to-gray function. The following procedures can be used:

```
FREEZE_DAC      PROC    FAR
; Disable changes to Palette and DAC registers during BIOS mode
; set operations
        MOV     AH,18       ; Alternate select BIOS service
        MOV     BL,49       ; Subservice to enable or disable
        MOV     AL,1        ; Code to disable changes
        INT     10H
        RET
FREEZE_DAC      ENDP
;
THAW_DAC        PROC    FAR
; Enable changes to Palette and DAC registers during BIOS mode
; set operations
        MOV     AH,18       ; Alternate select BIOS service
        MOV     BL,49       ; Subservice to enable or disable
        MOV     AL,0        ; Code to enable
        INT     10H
        RET
THAW_DAC        ENDP
```

## 6.3.2 16-Color Modes

In Table 5.1 we can see that VGA color modes can be in 2, 4, 16, and 256 colors. The two- and four-color modes are provided for compatibility with standards that are now obsolete; therefore they are of little interest to the present day VGA programmer. The same can be said of the lower-resolution graphics modes. This leaves us with 256-color modes, previously discussed, and with the 16-color graphics mode 18.

Video memory mapping in mode 18 can be seen in Figure 6.1, however, this illustration does not show how the color is obtained. Refer to Figure 5.3 to visualize how the pixel color in mode 18 is determined by the values stored in four maps, usually named intensity, red, green, and blue. This four-bit IRGB code is the number of one of 16 palette registers located in the Attribute Controller group. Furthermore, the value stored in the Palette register is also an address into the corresponding DAC Color register. This dual-level, indirect addressing scheme was developed to provide VGA compatibility with the CGA and the EGA standards. The matter is further complicated by the fact that the number of the DAC Color register (an 8-bit value in the range 0 to 255) can be stored differently. If the Palette Select bit of the Attribute Mode Control register is clear, then the number of the DAC Color register is stored in the six bits of the Palette register and in bits 2 and 3 of the Color Select register. If the Palette Select bit is set, then the number of the DAC Color register is stored in the four low-order bits of the Palette register and in the four low-order bits of the Color Select register. The two addressing modes are shown in Figure 6.7.

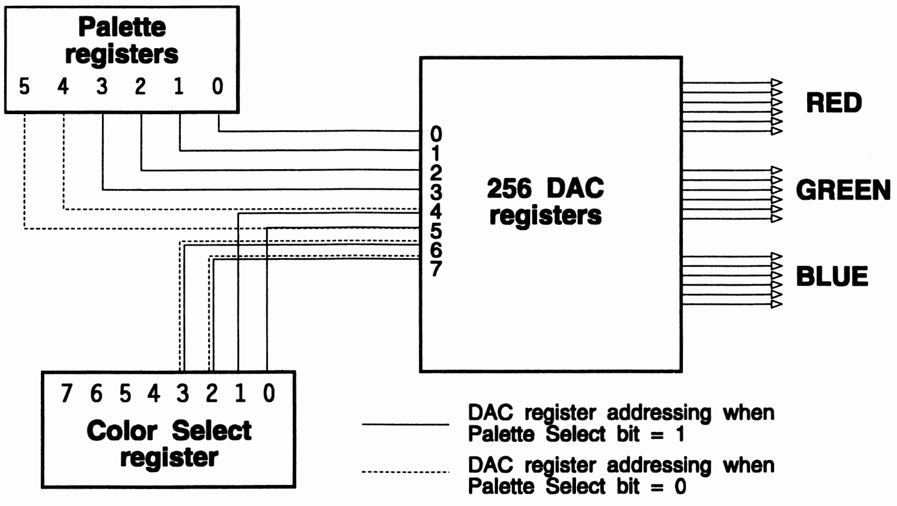


Figure 6.7 *Selection Modes for Active DAC Registers*

Notice in Figure 6.7 that when the Palette Select bit is set, bits 4 and 5 of the DAC register address are determined by bits 0 and 1 of the Color Select register, and not by bits 4 and 5 of the Palette register. A program operating in this addressing mode has to manipulate bits 4 and 5 of the desired DAC register number so that they are determined by bits 0 and 1 of the Color Select register, while bits 6 and 7 of the address are determined by bits 3 and 2 of the Color Select register.

The simplest and most straightforward color option for VGA mode 18 is to set the Palette Select bit and to clear bits 0 to 3 of the Color Select register. This mode of operation makes the Palette and Color Select registers transparent to the software. The DAC register number is now determined by the four low bits of the Palette register, which, in turn, match the IRGB value in the bit planes. The only disadvantage in this setup is that it is incompatible with the one in the CGA and EGA standards, which are based on the value stored in the 16 Palette registers. The method followed by the BIOS, designed to achieve compatibility with the Palette registers of the CGA and EGA cards, is based on a customized set of values for the DAC Color registers which are loaded during mode 18 initialization. This set, which includes values for the first 64 DAC Color registers only, can be seen in Table 6.6.

Table 6.6 *BIOS Settings for DAC Registers in Mode 18*

| No. | R | G | B | No. | R | G | B | No. | R | G | B | No. | R | G | B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 16 | 0 | 21 | 0 | 32 | 21 | 0 | 0 | 48 | 21 | 21 | 0 |
| 1 | 0 | 0 | 42 | 17 | 0 | 21 | 42 | 33 | 21 | 0 | 42 | 49 | 21 | 21 | 42 |
| 2 | 0 | 42 | 0 | 18 | 0 | 63 | 0 | 34 | 21 | 42 | 0 | 50 | 21 | 63 | 0 |
| 3 | 0 | 42 | 42 | 19 | 0 | 63 | 42 | 35 | 21 | 42 | 42 | 51 | 21 | 63 | 42 |
| 4 | 42 | 0 | 0 | 20 | 42 | 21 | 0 | 36 | 63 | 0 | 0 | 52 | 63 | 21 | 0 |
| 5 | 42 | 0 | 42 | 21 | 42 | 21 | 42 | 37 | 63 | 0 | 42 | 53 | 63 | 21 | 42 |
| 6 | 42 | 42 | 0 | 22 | 42 | 63 | 0 | 38 | 63 | 42 | 0 | 54 | 63 | 63 | 0 |
| 7 | 42 | 42 | 42 | 23 | 42 | 63 | 42 | 39 | 63 | 42 | 42 | 55 | 63 | 63 | 42 |
| 8 | 0 | 0 | 21 | 24 | 0 | 21 | 21 | 40 | 21 | 0 | 21 | 56 | 21 | 21 | 21 |
| 9 | 0 | 0 | 63 | 25 | 0 | 21 | 63 | 41 | 21 | 0 | 63 | 57 | 21 | 21 | 63 |
| 10 | 0 | 42 | 21 | 26 | 0 | 63 | 21 | 42 | 21 | 42 | 21 | 58 | 21 | 63 | 21 |
| 11 | 0 | 42 | 63 | 27 | 0 | 63 | 63 | 43 | 21 | 42 | 63 | 59 | 21 | 63 | 63 |
| 12 | 42 | 0 | 21 | 28 | 42 | 21 | 21 | 44 | 63 | 0 | 21 | 60 | 63 | 21 | 21 |
| 13 | 42 | 0 | 63 | 29 | 42 | 21 | 63 | 45 | 63 | 0 | 63 | 61 | 63 | 21 | 63 |
| 14 | 42 | 42 | 21 | 30 | 42 | 63 | 21 | 46 | 63 | 42 | 21 | 62 | 63 | 63 | 21 |
| 15 | 42 | 42 | 63 | 31 | 42 | 63 | 63 | 47 | 63 | 42 | 63 | 63 | 63 | 63 | 63 |

## 6.4 Color Animation

An interesting programming technique for VGA systems is to use the bits in the Color Select register to change some or all of the displayed colors. For example, if the Palette Select bit of the Attribute Mode Control register is clear, then bits 2 and 3 of the Color Select register provide two high-order bits of the DAC register number. Since two bits can encode four combinations (00, 01, 10, and 11), a program can change the value of bits 2 and 3 of the Color Select register to index into four separate areas of the DAC, each one containing 64 different color registers. By the same token, if the Palette Select bit is set, then the four low-order bits in the Color Select register can be used to choose one of

16 DAC areas, each one containing 16 color registers. The areas of the DAC determined through the Color Select register are sometimes referred to as *color pages*. Some interesting animation effects can be achieved by rapidly shifting these color pages. For example, a program can simulate an explosion by shifting the pixel colors to tints of red, orange, and yellow.

BIOS service number 16, subservice 19, provides a means for setting the paging mode to four color pages of 64 registers or to 16 color pages of 16 registers each and also for selecting an individual color page within the DAC. In this kind of programming it is important to remember that the BIOS initialization routines for mode 18 set color values for the first 64 DAC registers only. It is up to the software to initialize the color values in the other DAC registers.

Chapter

7

# VGA Mode X Drivers and Primitives

## 7.0 A Nonstandard VGA Mode

In Chapter 5 we mentioned that VGA graphics programmers have found ways of initializing the VGA hardware in order to create video modes different than those officially sponsored by IBM or other manufacturers. These are the so-called nonstandard modes. The best known of the VGA nonstandard modes is mode X. It appears that the mode X designation was first used by Michael Abrash in his column "Graphics Programming" that appeared in the July 1991 edition of *Dr. Dobb's Journal* (see Bibliography). The original article, titled "Mode X: 256-Color VGA Magic," continued in the August and September 1991 editions of the magazine. In his article Abrash mentions that the new mode is based on public domain code by John Bridges.

VGA mode X has a resolution of 320-by-240 pixels in 256 colors. In Table 5.1 we see that this resolution exceeds that of VGA mode 19 by 40 horizontal pixel rows. Notice that VGA mode 19 consists of 64,000 pixels (320 times 200), which is close to the maximum of 65,535 pixels that can be stored in a single segment or video map. In order to expand the number of pixels to 76,800 (320 times 240), the resolution of VGA mode X, it was necessary to adopt a planar scheme similar to the one in VGA mode 18.

For this reason the programmer finds that mode X is reminiscent of VGA mode 19 regarding the number of colors and screen dimensions and of mode 18 regarding the planar mapping of the video data. However, mode X is unique and differs in many ways from any of the standard modes. The programming methods and techniques for use in mode X are also quite different from those used in VGA standard modes. In fact, the VGA system operating in any particular graphics mode, standard or nonstandard, should be considered as a unique device. To the programmer there is often as much difference between VGA modes as between the VGA and other video systems.

### 7.0.1 Mode X Characteristics

Although VGA mode X falls considerably short of being the PC animator's panacea, it does offer some useful and interesting characteristics. Comparing VGA mode X to the standard VGA modes requires considering three separate factors: resolution, color range, and graphics performance.

Regarding resolution, mode X falls considerably short of VGA mode 18 and exceeds VGA mode 19. Mode 18 is capable of 640-by-480 pixels, making a total of 307,200 screen pixels, while mode 19 is capable of 320-by-200, for a total of 64,000 pixels. Mode X's resolution is 320-by-240, which makes a total of 76,800 pixels. VGA applications developers concerned with obtaining the highest possible level of screen detail usually prefer VGA mode 18 over mode X although, by skillfully applying dithering and antialiasing techniques it is possible to compensate for the lower resolution of mode X. In this manner a program designer can manipulate the greater color range of mode X to increase the apparent degree of object detail.

Mode X matches the best available color range in the VGA standard modes, which is 256 colors. As in mode 19, the color is encoded in one memory byte per pixel. Color mapping is by means of the DAC registers as described in Section 6.3.1. The same processing used for manipulating the DAC registers in mode 19 applies to mode X. This means a program designer interested in obtaining the highest possible color range in the VGA standard would probably select either mode 19 or mode X.

The element of graphics performance in the various VGA modes is difficult to evaluate exactly since many undeterminable elements enter into the equation. For example, a certain VGA mode has better performance than another one regarding operations on geometrical graphics objects and worse performance in the manipulation of objects defined in bitmaps. At the same time, a VGA planar mode (such as mode X) has much better performance when dealing with objects that are defined in a way that allows manipulating all or several video maps in parallel. Furthermore, at the system level, performance of a video mode is related to the absolute speed of the central processor, as well as its speed when accessing system memory and when accessing video memory space.

In spite of these complications it can be stated that, under typical processing conditions, the pixel-by-pixel performance of mode X code is considerably better than that of VGA mode 18 and somewhat better than that of VGA mode 19. The reasons for the exceptional performance of mode X are related to its peculiar architecture, as is shown in the following sections. In recent years animated programs that take advantage of mode X have thrived commercially, as well as in the Shareware and public domain fields.

### 7.0.2 Mode X in Animation Programming

Graphics programmers in general, and animators in particular, have found several attractive features in mode X. The following are the most important:

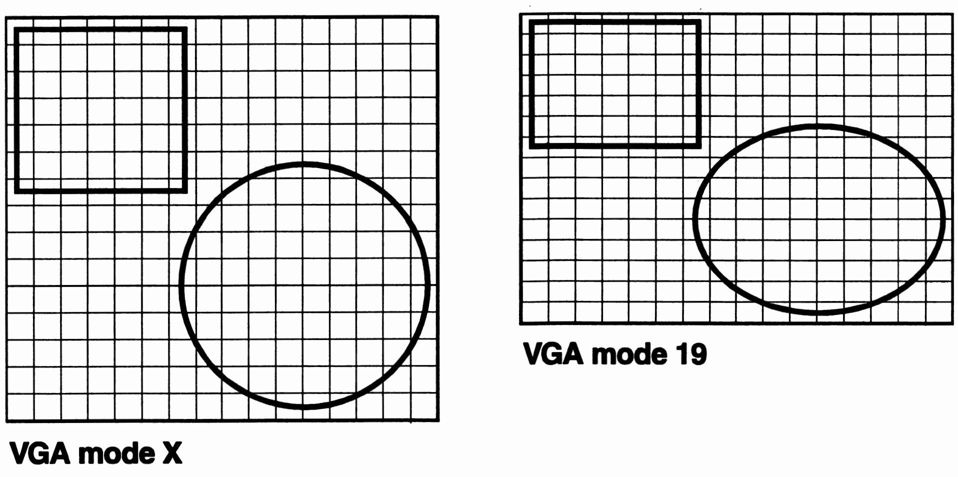


Figure 7.1 *Symmetrical and Asymmetrical Pixel Grids*

1. Mode X resolution is better than mode 19, the only other VGA mode in 256 colors.
2. Mode X operates on a symmetrical pixel grid. In other words, the VGA screen in mode X has a 1:1 aspect ratio. A rectangle of 20-by-20 pixels appears on the mode X screen as a square. In VGA mode 19, which has a nonsymmetrical grid, this rectangle does not appear square. The effect is shown in Figure 7.1. The VGA programmer working in a nonsymmetrical mode has to make compensations accordingly. In order to display a circle the application has to plot an ellipse, and a rectangle in order to display a square. The programmer working in a symmetrical mode (such as mode 18 or mode X) need not be concerned with these complications.
3. Mode X architecture is planar. Code can take advantage of the planar property of mode X by simultaneously reading and writing up to four pixels. Although this type of parallel processing is not applicable in all cases, programs can profitably use it in speeding up the performance of many graphics functions.
4. Because of its planar architecture, mode X operates on a folded memory space that totals 262,144 bytes. On the other hand, the linear memory space in mode 19 is 65,536 bytes, which is one-fourth of that in mode X. In practice, the memory space in mode X allows storing data for approximately 3.4 screens. This allows the use of off-screen video memory for storing images that can be rapidly moved to the displayed area and for a page flipping technique much used in animation. In this chapter we develop several processing routines that take advantage of mode X's off-screen video memory space.

5. Color mapping in mode X is on a byte-per-pixel basis. This method makes unnecessary the complicated bit masking operations required in VGA mode 18, in which each screen pixel is mapped to a video memory bit.

On the other hand, mode X also has disadvantages. One of them is that the planar architecture complicates programming since each screen pixel is stored in one of four maps. The code must perform several operations in order to determine and access the corresponding map. This additional processing load affects the performance of some graphics functions when compared to the direct memory-to-video mapping of mode 19.

The program designer must carefully analyze the possibilities and limitations of each VGA mode in order to determine the one that is most convenient for a particular application. Mode 18 shows the best resolution but not the most extensive color range. Mode 19 has the maximum color range and is easy to program, but it has very little off-screen video memory, a nonsymmetrical pixel grid, and low resolution. Mode X has the same color range as mode 19, a symmetrical pixel grid and plenty of off-screen memory, but its planar architecture makes some graphics operations slower than in mode 19, programming is considerably more difficult, and resolution is much lower than in mode 18.

## 7.1 VGA Mode X Architecture

The principal characteristic of VGA mode X architecture is its unique planar configuration, whereby consecutive screen pixels are stored in different memory maps. Figure 7.2 shows four adjacent screen pixels, each one stored in a different memory map.

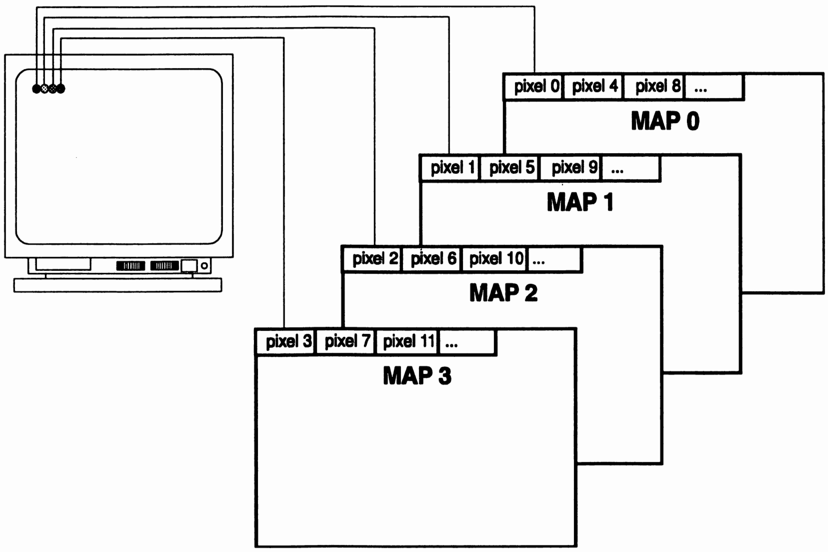


Figure 7.2 *Video Memory Mapping in VGA Mode X*

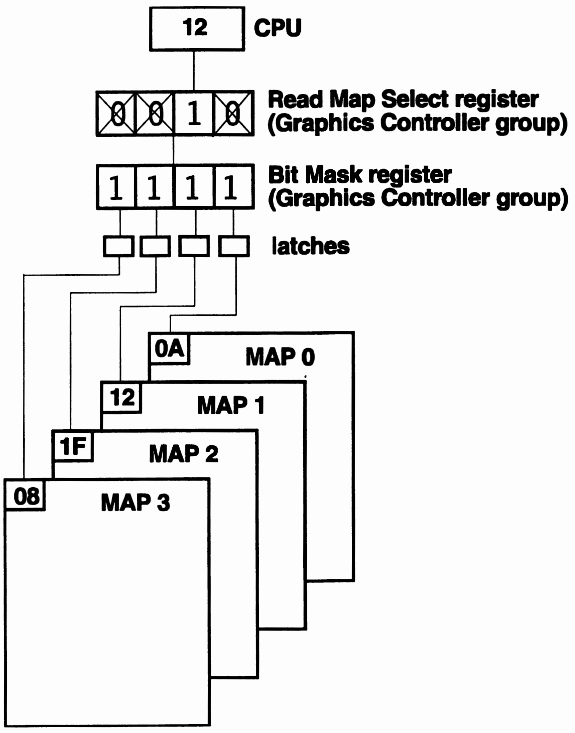

12
CPU
0 0 1 0
Read Map Select register (Graphics Controller group)
1 1 1 1
Bit Mask register (Graphics Controller group)
latches
0A
MAP 0
12
MAP 1
1F
MAP 2
08
MAP 3

Figure 7.3 *Mode X Video Map Access During Read*

What makes this feature of VGA mode X unique, and somewhat confusing, is that all four screen pixels are mapped to the same physical address. Therefore, the first four screen pixels, located at the top-left screen corner, are accessed at memory byte A0000H. Which of the four pixels at this location is accessed by the CPU depends on the setting of the corresponding VGA hardware registers. During read operations the Read Map Select register of the Graphics Controller group determines which of the four maps is accessed by the processor. Figure 7.3 shows the contents of map 1 being accessed by the CPU.

Since the screen in VGA mode X contains 320 pixel columns by 240 pixel rows, the total number of screen pixels is of 76,800. However, since four pixels are stored per physical address in video memory, the video screen is mapped to 19,200 addresses in the buffer. In other words, the planar mapping mechanism of mode X allows storing 76,800 pixels in 19,200 addresses, each of which is associated with one of four maps or planes.

According to Figure 7.3, in order to read the contents of four adjacent screen pixels, the code must successively enable each of the four maps by means of the Read Map Select register. Consequently, to read and store the contents of the entire screen in VGA mode X, the code performs four read operations for each four adjacent pixels.

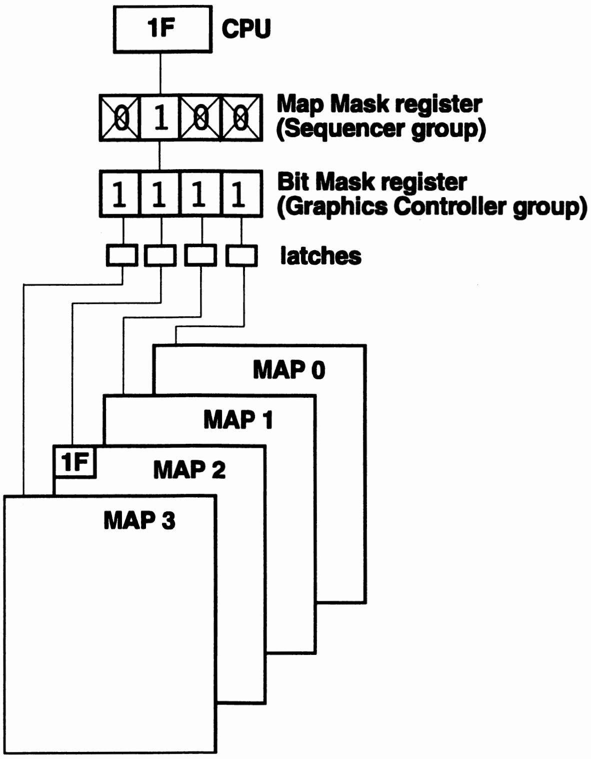


Figure 7.4 *Mode X Video Map Access During Write*

Mode X operations are best visualized by introducing the concept of latches. A *latch* is a temporary storage space that is loaded with the contents of a memory map during a read operation. Which map is "latched" onto the CPU is determined by the setting of the Read Map Select register, as shown in Figure 7.3. The Bit Mask register of the Graphics Controller determines the map to which the CPU has access. In order to provide access to all four maps the Bit Mask register should be loaded with all 1 bits.

Write operations require setting the Map Mask register of the Sequencer to determine which memory map is accessed with the CPU data. In this case the Bit Mask register of the Graphics Controller also determines which maps are visible to the CPU. A 0 bit in the Bit Mask register makes the corresponding latch inoperative. Figure 7.4 shows a write operation in which the CPU contents are stored in video map 2.

At this point it could seem that the setting of the Bit Mask register is trivial during mode X read and write operations, since in Figures 7.3 and 7.4 the Bit Mask is loaded with 1 bits and CPU access is controlled by means of either the Map Mask or the Read Map Select registers. This is not entirely the case, since the Bit Mask register can be used to provide parallel access to several planes during data transfers within video memory. Code can set all 0 bits in the Bit Mask register, perform a read operation to load the four latches, and then store

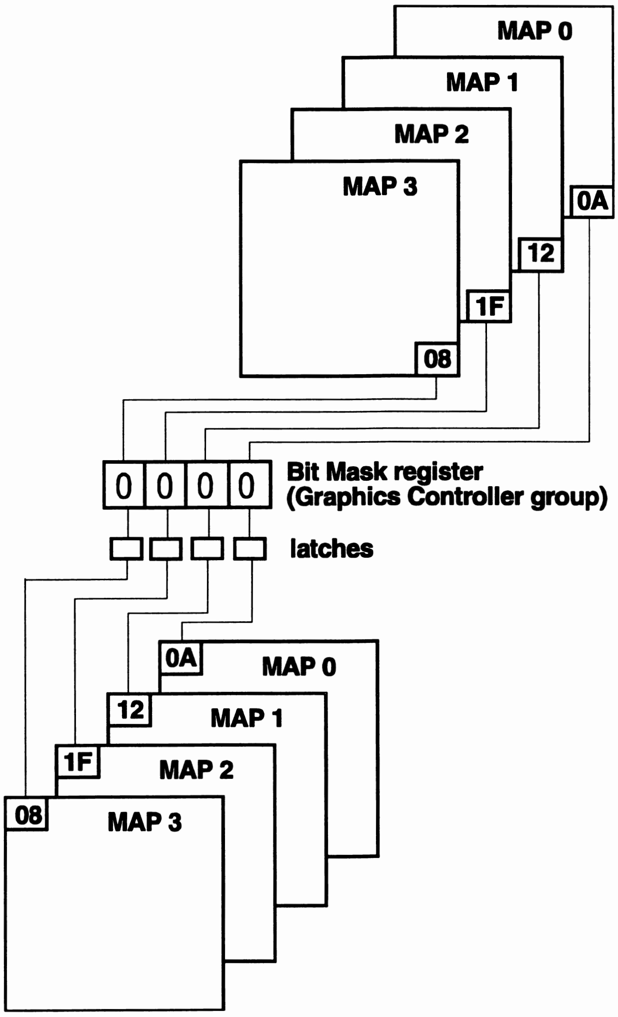


Figure 7.5 *Mode X Parallel Map Access*

the contents of the latches with a single write operation. In this case four bytes of data are transferred with a single access, thus increasing performance considerably. Figure 7.5 is a representation of a 4-byte transfer of video data using the setting of the Bit Mask register.

### 7.1.1 Pixel-Level Address Calculations

Address calculations in mode X consist in determining the pixel's offset from the start of the video buffer, as well as the pixel's position within the 4-pixel group located at the same physical location. Figure 7.6 shows a pixel located in column 70, row 82, of the VGA mode X video screen.

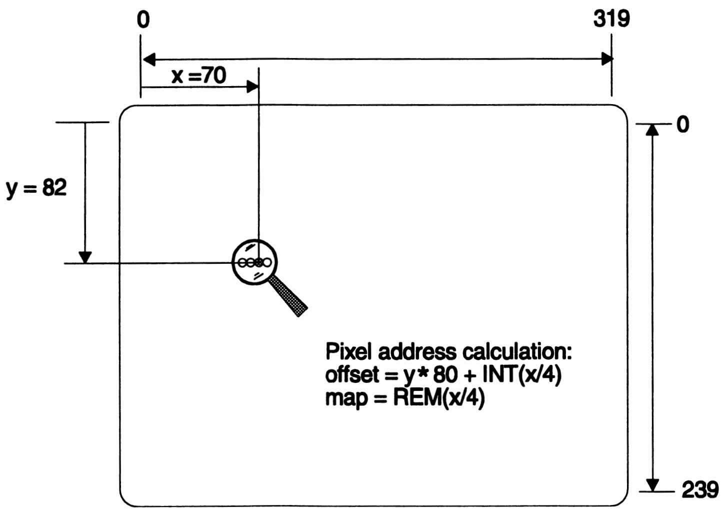


Figure 7.6 *Example of Mode X Pixel Address Calculation*

Pixel address calculation in VGA mode X requires multiplying the pixel row by 80, which is the number of video buffer bytes in each row, and then dividing the pixel column by 4, which is the number of pixels mapped to each buffer storage location. The pixel's location within the 4-pixel group is the remainder from this division. In reference to Figure 7.6, we perform the following operations:

```
        82 * 80 = 6560 (row offset)
  + INT(70 / 4) =   17 (column offset)
                -----
                  6577 (pixel offset)
      REM(70/4) = 2 (pixel position in 4-pixel group)
```

In Figure 7.2 we see that in VGA mode X the video map 0 corresponds to the left-most pixel in the group and video map 3 to the right-most pixel. Therefore, the remainder that results from dividing the pixel column by 4 is equal to the number of the video map in which the pixel is located. In the example of Figure 7.6 the pixel's offset from the start of the video buffer is 6577 and the pixel is located in map number 2. The following code fragment performs the pixel address calculations in VGA mode X:

```
; Code to calculate pixel address and map number in VGA mode X
;          CX = x coordinate of pixel (range 0 to 319)
;          DX = y coordinate of pixel (range 0 to 239)
;***************************|
; calculate buffer address  |
;***************************|
        PUSH    CX          ; Save x coordinate
```

```
; Formula: offset = (y * 80) + (x/4)
        MOV     AX,DX           ; y coordinate to AX
        MOV     CL,80           ; Multiplier to CL
        MUL     CL              ; AX = y * 80
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
; Prepare for division
        MOV     CL,4            ; Divisor to CL
        DIV     CL              ; AX / 4  = quotient in AL and
                                ; remainder in AH
        MOV     CL,AH           ; Store remainder in CL (map number)
        MOV     AH,0            ; Clear remainder for addition
        ADD     BX,AX           ; Add into buffer offset
; At this point BX holds to video buffer offset and CL holds the
; number of the video map
        .
        .
        .
```

### 7.1.2 Tile-Level Address Calculations

In Chapter 6 we mentioned that graphics software can often profit from the convenience and greater performance of routines that access the video display at a larger-than-pixel level. The architecture of VGA mode X allows simultaneously setting four screen pixels to the same color by manipulating the Map Mask register of the Sequencer and the Bit Mask register of the Graphics Controller. Figure 7.7 shows how the value stored in the CPU is simultaneously copied to the four video maps.

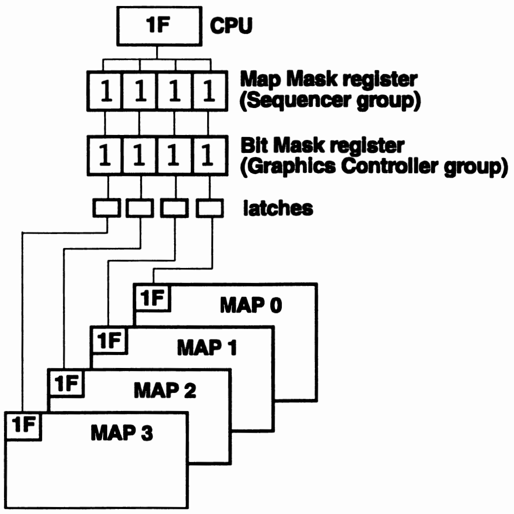


Figure 7.7 *Simultaneous Map Write Operation in VGA Mode X*

We can visualize the video display surface in VGA mode X as formed by 4-by-4 pixel tiles, both on the horizontal and in the vertical planes. A routine can then be devised that accesses the display surface in groups of four pixels (as shown in Figure 7.7). This routine would set a screen tile (a 16-pixel area) with four write operations, one for each group of four horizontal pixels. The improved performance of this approach could be convenient in those graphics functions in which this lower resolution is acceptable, for example, in clearing the screen or in displaying rectangles delimited at the screen tile level.

Tile address calculations in mode X consist in determining the tile's offset from the start of the video buffer. Notice that each tile row consists of four pixel rows, each one represented in 80 buffer addresses. Therefore, there are a total of 320 pixels in each tile row. Figure 7.8 shows a tile located in column 20, row 39, of the VGA mode X video screen.

Tile address calculation in VGA mode X requires multiplying the tile row by 320, which is the number of video buffer bytes in each tile row, and then adding the tile column. In reference to Figure 7.8, we perform the following operations:

```
 20 * 320 = 6400 (row offset)
+             39 (column offset)
          ------
            6439 (tile offset)
```

In the example of Figure 7.8 the tile's offset from the start of the video buffer is 6439. The following code fragment performs the tile address calculations in VGA mode X:

```
; Code to calculate tile address in VGA mode X
;       CL = horizontal tile number (range 0 to 79) (x coordinate)
```

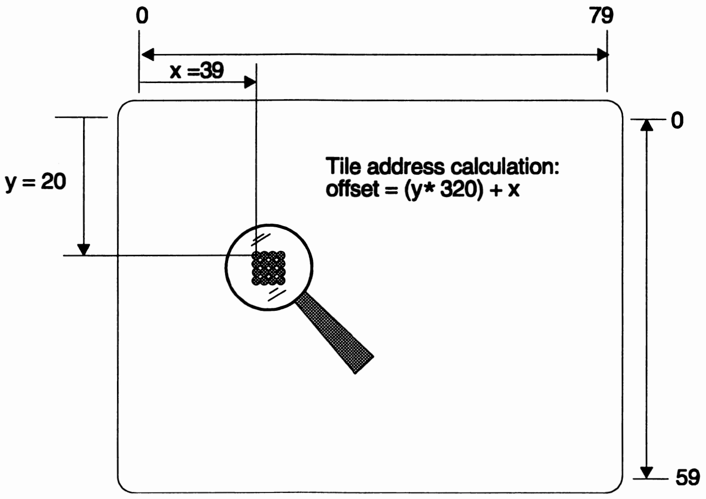


Figure 7.8 *Example of Mode X Tile Address Calculation*

```
;       CH = vertical tile number (range 0 to 59) (y coordinate)
; Compute coarse-grain address (in BX) as follows:
;          BX = (CH * 320) + CL
;
        PUSH    CX              ; Save x and y tile coordinates
; Calculate tile address
        MOV     BX,CX           ; Copy CX in BX
        MOV     BH,0            ; Clear x and leave y coordinate
; Add in y coordinate times 320
        POP     AX              ; Restore entry tile number (AL)
        MOV     AL,AH           ; Transfer column to AL
        MOV     AH,0            ; Clear high byte
        MOV     CX,320          ; Multiplier
        MUL     CX              ; AX = AX * 320
        ADD     BX,AX           ; Add in tile offset
; At this point BX holds the tile offset
```

### 7.1.3 The Video Buffer in Mode X

In Section 7.1 we saw how the planar architecture of mode X makes possible compressing 4 pixels into a single storage address. Since the VGA system assigns 64K (65,536 bytes) to the physical mapping of the video space, a 4:1 compression allows storing the state of 262,143 pixels in the assigned space. Considering that the resolution of mode X is of 320-by-240 pixels, each full screen contains 76,800 pixels and takes up 19,200 physical address locations. Figure 7.9 shows the distribution of the physical video buffer space into video pages.

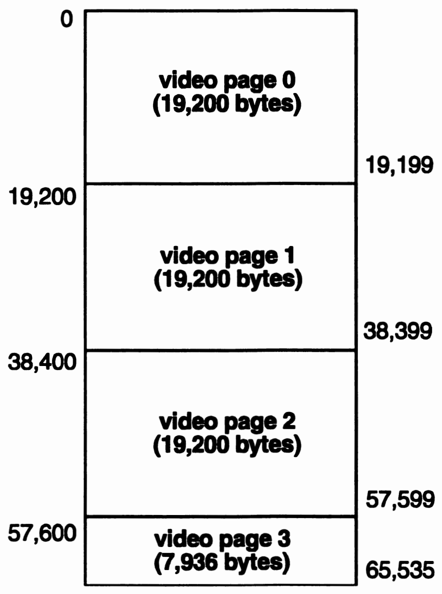


Figure 7.9 *Video Buffer Space in VGA Mode X*

Notice in Figure 7.9 that video page 3 is not a full page. The programmer must manage video buffer space carefully, since overflowing the available area usually produces a system crash.

The availability of off-screen buffer storage is one of the most useful features of VGA mode X; therefore, it is usually a good idea to include video paging into the address calculation routines for this mode. Since each video page contains 19,200 locations in the video buffer, the address calculation routine can multiply this constant by the page number, which must be in the range 0 to 3, and add the product to the offset as calculated in the code samples of Sections 7.1.1 and 7.1.2. The VGA mode X procedures listed in this chapter take into account the video page when calculating video buffer addresses.

## 7.2 Setting Mode X

Because mode X is not a standard VGA mode, it cannot be set using a BIOS service. However, since mode X resembles VGA standard mode 19, it is possible to save considerable programming effort by letting the BIOS set mode 19 and then making the adjustments in the VGA hardware that are necessary to mode X. The following procedure uses this method to set mode X. The code is based on the one listed by Michael Abrash in the reference previously noted. We have streamlined the processing to make it more suited to assembly language, hard-coded some of the equates to improve readability, and added code to turn off the VGA display function while resetting the hardware in order to avoid screen garbage.

```
;****************************************************************
;                         set VGA mode X
;****************************************************************
; Code segment data for mode X setting
; Each entry in the following table contains the offset of the
; corresponding data register of the CRT Controller and the
; data byte to be installed. The low-order byte is the register
; offset and the high-order byte the data
; The values are those required to change mode 19 to mode X
CRT_DATA     DW    00D06H  ; Vertical Total register (8 low-order
                           ; bits of 10-bit value
             DW    03E07H  ; Overflow register (bit 5 is VT 9 bit
                           ; and bit 0 is VT 8 bit)
             DW    04109H  ; Maximum Scan Line register
             DW    0EA10H  ; Vertical Retrace Start register
             DW    0AC11H  ; Vertical Retrace End
             DW    0DF12H  ; Vertical Display-Enable End
             DW    00014H  ; Underline Location and Doubleword
             DW    0E715H  ; Start Vertical Blanking
             DW    00616H  ; End Vertical Blanking
             DW    0E317H  ; CRT Mode Control
; Length is 10 doubleword items
```

```
SET_MODE_X      PROC    FAR
; Procedure to set VGA nonstandard mode X
; Code uses BIOS service to set mode 19 (320-by-200 pixels in
; 256 colors) and then makes the necessary adjustments
;
; Use BIOS subservice to set mode 19
        MOV     AL,19   ; Mode number
        MOV     AH,0    ; BIOS subservice number
        INT     10H
;***************************|
;     turn off video        |
;***************************|
; To avoid screen garbage the VGA functions are turned off while
; setting the mode X registers
        MOV     AX,1201H        ; Service number and OFF code
        MOV     BL,36H          ; subservice request
        INT     10H
;***************************|
;   modify the VGA hardware |
;***************************|
; Make changes in Sequencer registers
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0604H ; Memory Mode register as follows:
                ; 0 0 0 0 0 1 1 0
                ;           | | |____ Extended memory ON
                ;           | |______ Odd/even maps to ODD
                ;           |________ Chain maps disabled
        OUT     DX,AX
; Access Sequencer Reset register
        MOV     AX,0100H ; Reset register as follows:
                ; 0 0 0 0 0 0 0 1
                ;             | |____ ASR bit ON
                ;             |______ SR bit OFF
        OUT     DX,AX
; Access Miscellaneous Output of the General register
        MOV     DX,3C2H  ; Miscellaneous Output register
        MOV     AL,0E3H  ; Reset register as follows:
                ; 1 1 1 0 0 0 1 1
                ; | | |   | | | |____ I/O select to 3DxH
                ; | | |   | | |______ Enable RAM decoding
                ; | | |   |_|________ Clock select to 25.175 MHz
                ; | | |______________ Page select to EVEN
                ; |_|________________ 480 lines vertical size
        OUT     DX,AL
; Access Reset register to restart Sequencer
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0300H ; Reset register as follows:
                ; 0 0 0 0 0 0 1 1
                ;             | |____ ASR bit ON
                ;             |______ SR bit ON
        OUT     DX,AX
```

```
; Access Vertical Retrace End register of the CRT Controller
; (CRT Controller is at port 3D4H)
        MOV     DX,3D4H  ; CRT Controller base address
        MOV     AL,11H   ; Address of Vertical Retrace End
        OUT     DX,AL    ; Select this Data register
        INC     DX       ; Point to Data registers
        IN      AL,DX    ; Read Vertical Retrace End
        AND     AL,7FH   ; Clear bit 7 (Protect registers 0-7)
        OUT     DX,AL
                ; 0 0 0 0 0 0 0 1
                ;              | |____ ASR bit ON
                ;              |______ SR bit OFF
; Prepare to output data block to CRT Controller registers
        DEC     DX      ; CRT Controller base address
        LEA     SI,CS:CRT_DATA  ; Set pointer to data table
        MOV     CX,10   ; Number of entries in table
; Output data block
SET_CRTC:
        MOV     AX,CS:[SI]      ; Load data pair into AX
        OUT     DX,AX   ; Select register and write data
        INC     SI      ; Bump pointer
        INC     SI      ; twice
        LOOP    SET_CRTC
;***************************|
;      clear display        |
;***************************|
; Access Map Mask register of the Sequencer
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0F02H ; Reset register as follows:
                ; 0 0 0 0 1 1 1 1
                ;         |_|_|_|_____ Enable all 4 maps
        OUT     DX,AX
; Set ES to video segment base
        MOV     AX,0A000H ; Video memory segment base
        MOV     ES,AX     ; To ES
; Set pointers, counter, and data bytes
        XOR     DI,DI     ; ES:DI — start of video buffer
        XOR     AX,AX     ; Pixel data to write is 00 00
        MOV     CX,8000H  ; Counter for total video memory words
; Clear memory
        CLD               ; Direction is forward
        REP     STOSW     ; Store to clear memory
; On exit ES — base address of video buffer
;***************************|
;     turn on video         |
;***************************|
        MOV     AX,1200H        ; Service number and ON code
        MOV     BL,36H          ; Subservice request
        INT     10H
        RET
SET_MODE_X      ENDP
```

## 7.3 Pixel-Level Device Drivers

Although pixel-by-pixel operations in VGA mode X have a low performance level, most graphics applications require this degree of control. The fundamental pixel-level device drivers consist of a routine to set an individual screen pixel and another one to read a screen pixel.

### 7.3.1 VGA Mode X Write Pixel Procedure

The following procedure performs the necessary operations for setting a single screen pixel. It uses code segment constants and variables, listed in a header area. These parameters are also used by the procedure to read a screen pixel listed in Section 7.3.2.

```
;****************************************************************
;     code segment variables and constants for pixel-level
;                      mode X device drivers
;****************************************************************
VIDEO_PAGE  DW      0       ; Word storage for video page
; Code segment variables for mode X address calculations
EIGHTY      DW      80      ; Pixels per row
FOUR        DB       4      ; Number of planes
;
WRITE_PIX_X     PROC    FAR
; Set screen pixel while in VGA mode X
; On entry:
;          CX = x coordinate of pixel (range 0 to 319)
;          DX = y coordinate of pixel (range 0 to 239)
;          AL = pixel color
;          AH = display page (range 0 to 3)
;
;**************************|
;   save caller's context   |
;**************************|
        PUSH    DX              ; Save y coordinate
        PUSH    CX              ; Save x coordinate
        PUSH    AX              ; Save pixel color
        PUSH    CX              ; Save x coordinate
; Store page number
        MOV     AL,AH           ; Page number to AL
        MOV     AH,0            ; Clear high byte
        MOV     CS:VIDEO_PAGE,AX ; Store it
;
;**************************|
; calculate buffer address  |
;**************************|
; Formula: offset = (y * 80) + (x/4)
        MOV     AX,DX           ; y coordinate to AX
        MUL     CS:EIGHTY       ; AX = y * 80
        MOV     BX,AX        ; Free AX and hold in BX
        POP     AX           ; x coordinate from stack
; Prepare for division
        DIV     CS:FOUR      ; AX / 4  = quotient in AL and
                             ; remainder in AH
        MOV     AH,0         ; Clear remainder for addition
        ADD     BX,AX        ; Add into buffer offset
; Calculate and add video page offset
        MOV     AX,19200     ; Length of each page in mode X
        MUL     CS:VIDEO_PAGE ; Multiply by page number
        ADD     BX,AX        ; Add into offset
; Determine pixel plane of address by masking off bits 2-7
; At this point CX = x coordinate of pixel address
        AND     CL,00000011B    ; Mask off bits
;
;**************************|
;    select active map     |
;**************************|
; Select active map by setting the Sequencer's Map Mask
; register
        MOV     DX,3C4H      ; Sequencer base address
        MOV     AX,0102H     ; AL = Map Mask register offset
                             ; AH = 0000 0001
                             ;              |___ map 0 enabled
; Shift enable bit to left by plane number (in CL)
        SHL     AH,CL        ; If CL = 0 then map 0 enabled
                             ; If CL = x then map x enabled
        OUT     DX,AX
;
;**************************|
; allow CPU access to maps |
;**************************|
; All bits in the Bit Mask register of the Graphics Controller
; must be set to 1 to allow CPU access
        MOV     DX,3CEH      ; Graphics Controller base address
        MOV     AX,0FF08H    ; AL = Bit Mask register
                             ; AH = 11111111B to load maps from
                             ; CPU
        OUT     DX,AX
;
;**************************|
;       set the pixel      |
;**************************|
        POP     AX           ; Restore pixel color from stack
; Code assumes that ES — video buffer segment base (A000H)
        MOV     ES:[BX],AL   ; Set pixel
        POP     CX           ; Restore pixel address registers
        POP     DX
        RET
WRITE_PIX_X     ENDP
```

### 7.3.2 VGA Mode X Read Pixel Procedure

The following procedure reads the color attribute of a single screen pixel while in VGA mode X. The procedure uses code segment constants listed in the header of the WRITE_PIX_X procedure in Section 7.3.1.

```
READ_PIX_X      PROC    FAR
; Read pixel while in VGA mode X
; On entry:
;          CX = x coordinate of pixel (range 0 to 319)
;          DX = y coordinate of pixel (range 0 to 239)
; On exit:
;          AL = 8-bit color of selected pixel
;          AH = page number (range is 0 to 3)
;**************************|
;   save caller's context   |
;**************************|
        PUSH    DX              ; Save y coordinate
        PUSH    CX              ; Save x coordinate
        PUSH    CX              ; Save x coordinate
; Store page number
        MOV     AL,AH           ; Page number to AL
        MOV     AH,0            ; Clear high byte
        MOV     CS:VIDEO_PAGE,AX; Store it
;**************************|
;  calculate pixel address  |
;**************************|
        MOV     AX,DX           ; y coordinate to AX
        MUL     CS:EIGHTY       ; AX = y times 80
        MOV     BX,AX           ; Free AX and hold in BX
        POP     AX              ; x coordinate from stack
        DIV     CS:FOUR         ; AX / 4  = quotient in AL and
                                ; remainder in AH
        MOV     AH,0            ; Clear remainder for addition
        ADD     BX,AX           ; Add into buffer offset
; Calculate and add video page offset
        MOV     AX,19200        ; Length of each page in mode X
        MUL     CS:VIDEO_PAGE ; Multiply by page number
        ADD     BX,AX           ; Add into offset
; Determine pixel plane of address by masking off bits 2-7
; At this point CX = x coordinate of pixel address
        AND     CL,00000011B    ; Mask off bits
        MOV     AH,CL           ; Pixel plane to AH
;**************************|
;     select video map      |
;**************************|
; Select active map by setting the Graphics Controller
; Read Map Select register
        MOV     DX,3CEH         ; Graphics Controller base address
        MOV     AL,04H          ; AL = Read Map Select register
```

```
                                ; AH = pixel plane to enable
        OUT     DX,AX
;***************************|
;      read the pixel        |
;***************************|
; Code assumes that ES — video buffer segment base
        MOV     AL,ES:[BX] ; Read selected map into AL
        POP     CX         ; Restore pixel address registers
        POP     DX
        RET
READ_PIX_X      ENDP
```

## 7.4 Tile-Level Device Drivers

It is possible to take advantage of the greater performance of parallel write operations in VGA mode X by accessing the display at a tile level. A tile-level device driver receives the tile coordinates, pixel color, and video display page from the caller and sets the corresponding group of 4-by-4 pixels. The performance gain results from setting four pixels at a time by manipulating the Map Mask register of the Sequencer and the Bit Mask register of the Graphics Controller (see Figure 7.7).

### 7.4.1 VGA Mode X Write Tile Procedure

The following procedure performs the necessary operations for setting a single screen tile. It uses code segment constants listed in a header area.

```
;****************************************************************
;      code segment variables and constants for tile-level
;                     mode X device drivers
;****************************************************************
VIDEO_PAGE      DW      0    ; Word storage for video page
; Code segment variables for mode X address calculations
THREE_20    DW     320       ; Factor for tile address
;****************************************************************
;                     write tile procedure
;****************************************************************
WRITE_TILE_X    PROC    FAR
; Procedure to calculate the coarse-grain address at a 4-by-4
; pixel grain and set tile, while in VGA mode X
;
; On entry:
;       CL = horizontal tile number (range 0 to 79) (x coordinate)
;       CH = vertical tile number (range 0 to 59) (y coordinate)
;       AL = pixel color
;       AH = video page (range is 0 to 3)
;
```

```
; Compute coarse-grain address (in BX) as follows:
;         BX = (CH * 320) + CL
;
;**************************|
;   save caller's context   |
;**************************|
        PUSH    CX          ; Save caller's CX
        PUSH    DX          ; Save caller's DX
        PUSH    AX          ; Save accumulator
        PUSH    CX          ; Save x and y tile coordinates
; Store page number
        MOV     AL,AH           ; Page number to AL
        MOV     AH,0            ; Clear high byte
        MOV     CS:VIDEO_PAGE,AX ; Store it
;**************************|
;   calculate tile address  |
;**************************|
        MOV     BX,CX       ; Copy CX in BX
        MOV     BH,0        ; Clear x and leave y coordinate
; Add in y coordinate times 320
        POP     AX          ; Restore entry tile number (AL)
        MOV     AL,AH       ; Transfer column to AL
        MOV     AH,0        ; Clear high byte
        MUL     CS:THREE_20 ; AX = AX * 320
        ADD     BX,AX       ; Add in tile offset
; Calculate and add video page offset
        MOV     AX,19200    ; Length of each page in mode X
        MUL     CS:VIDEO_PAGE ; Multiply by page number
        ADD     BX,AX       ; Add into offset
;**************************|
;   select all video maps   |
;**************************|
; Select all maps by setting all bits in the Sequencer's Map
; Mask register
        MOV     DX,3C4H     ; Sequencer base address
        MOV     AX,0F02H    ; AL = Map Mask register offset
                            ; AH = 0FH (all maps enabled)
        OUT     DX,AX
; All bits in the Bit Mask register of the Graphics Controller
; must be set to 1 to allow CPU access
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0FF08H; AL = Bit Mask register
                         ; AH = 11111111B to load maps from
                         ; CPU
        OUT     DX,AX
;**************************|
;        set tile           |
;**************************|
        POP     AX          ; Restore pixel color from stack
; Prepare to repeat 4 horizontal segments of 4 pixels each
        MOV     CX,4    ; Counter
```

```
; Code assumes that ES — video buffer segment base (A000H)
SET_TILE:
        MOV     ES:[BX],AL  ; Set pixel
        ADD     BX,80       ; Index to next pixel row
        LOOP    SET_TILE
        POP     DX          ; Restore caller's DX
        POP     CX          ; and CX
        RET
WRITE_TILE_X    ENDP
```

### 7.4.2 Setting Multiple Tiles

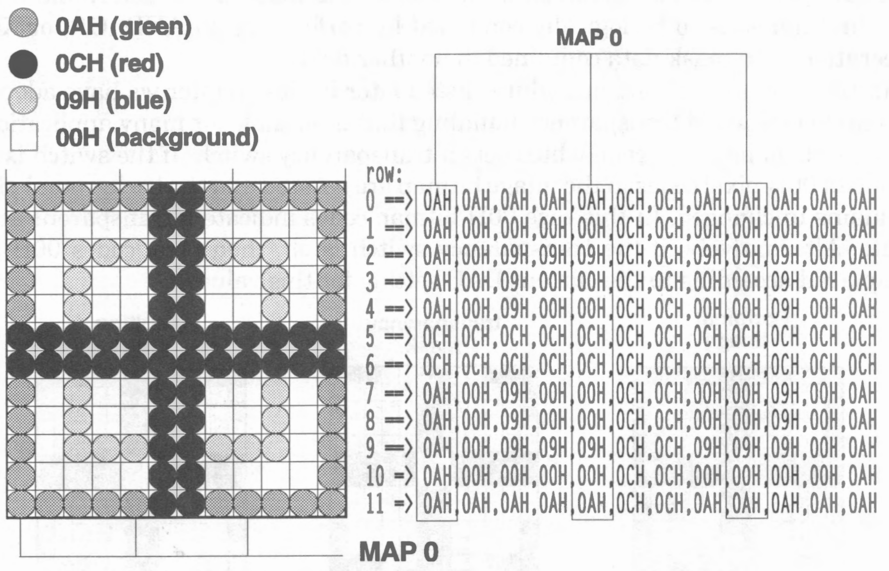
Code often requires to set a rectangle of adjacent screen tiles. The entire video screen can be described as one such rectangle. The super procedure for setting a tile group is based on the device driver listed in the previous section.

```
;****************************************************************
;                    write multiple tiles
;****************************************************************
; Code segment data for MULTI_TILE_X procedure
REPEAT_X        DB      0   ; Horizontal tile counter
MULTI_TILE_X    PROC    FAR
; Procedure to set multiple 4-by-4 pixel tiles in VGA mode X
;
; On entry:
;       CL = horizontal tile number (range 0 to 79) (x coordinate)
;       CH = vertical tile number (range 0 to 59) (y coordinate)
;       DL = horizontal tile count (minimum is 1)
;       DH = vertical tile count (minimum is 1)
;       AL = pixel color
;       AH = video page (range is 0 to 3)
;
;**************************|
; store rectangle dimensions|
;**************************|
        MOV     CS:REPEAT_X,DL   ; x repetitions
;**************************|
;    set tile rectangle    |
;**************************|
NEXT_TILE_ROW:
        PUSH    CX              ; Save start address
NEXT_TILE:
        CALL    FAR PTR WRITE_TILE_X     ; Set tile
        INC     CL               ; Index to next tile
        DEC     DL               ; Horizontal counter
        JNZ     NEXT_TILE        ; Continue
; At this point one horizontal tile line has been laid
        MOV     DL,CS:REPEAT_X   ; Reload variable
        POP     CX               ; Restore row address
        INC     CH               ; Next tile row
        DEC     DH                  ; Vertical counter
        JNZ     NEXT_TILE_ROW       ; Continue
        RET
MULTI_TILE_X    ENDP
```

## 7.5 VGA Mode X Bitmap Primitives

Although vector images offer many advantages, it is a rare PC graphics application that does not use bit-mapped images. The most unrefined and ineffective way of displaying or acquiring a bit-mapped image is by means of the procedures to set and read an individual screen pixel developed in Sections 7.3.1 and 7.3.2. The program designer can often take advantage of mode X architecture in creating bitmaps that can be manipulated much more effectively by the code.

The output routine to display a bit-mapped image in VGA mode X has a performance bottleneck in the hardware access instructions (OUT) that are necessary to select each one of the four video maps. If the code were to output the bitmap on a pixel-by-pixel basis, then the Map Mask register of the Sequencer would have to be accessed once for every pixel in the bitmap. Figure 7.10 shows an image and bitmap encoded in 4 colors. Since the image consists of 12 horizontal pixels and 12 pixel rows, in this case the code would have to access the Map Mask register 144 times, once for each pixel, if it were to be displayed pixel by pixel. Later in this chapter we develop VGA mode X procedures to display and acquire bitmaps in a much more effective manner.



```
row:
0 ==> 0AH,0AH,0AH,0AH,0AH,0CH,0CH,0AH,0AH,0AH,0AH,0AH
1 ==> 0AH,00H,00H,00H,00H,0CH,0CH,00H,00H,00H,00H,0AH
2 ==> 0AH,00H,09H,09H,09H,0CH,0CH,09H,09H,09H,00H,0AH
3 ==> 0AH,00H,09H,00H,00H,0CH,0CH,00H,00H,09H,00H,0AH
4 ==> 0AH,00H,09H,00H,00H,0CH,0CH,00H,00H,09H,00H,0AH
5 ==> 0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH
6 ==> 0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH,0CH
7 ==> 0AH,00H,09H,00H,00H,0CH,0CH,00H,00H,09H,00H,0AH
8 ==> 0AH,00H,09H,00H,00H,0CH,0CH,00H,00H,09H,00H,0AH
9 ==> 0AH,00H,09H,09H,09H,0CH,0CH,09H,09H,09H,00H,0AH
10 => 0AH,00H,00H,00H,00H,0CH,0CH,00H,00H,00H,00H,0AH
11 => 0AH,0AH,0AH,0AH,0AH,0CH,0CH,0AH,0AH,0AH,0AH,0AH
```

Figure 7.10 *Sample Image and Bitmap in VGA Mode X*

## 7.5.1 Pixel Transparency

Notice in Figure 7.10 that the image and bitmap pixels with the value 00H are described as having the same attribute as the background. In bitmap terminology these pixels are said to be transparent. Code can ignore pixel transparency or can handle it in one of several ways. When transparency is ignored, all image pixels are displayed according to the encoded attribute. In the case of the image in Figure 7.10 this would mean that the pixels with the value 00H would be displayed with the color code 00H, which is black in the default VGA DAC register setting for 256-color modes. Therefore if the image in Figure 7.10 were to be overlaid over a nonblack background, the pixels with code 00H would be displayed as black dots and the background would be completely obscured.

For many applications the nontransparent handling of bitmap attributes is not satisfactory. For example, if the image in Figure 7.10 were to be used as a graphics cursor to mark a certain position on the video display, then it would be preferable if the pixels that are not part of the cursor image remain with the background attributes. There are several ways of handling pixel transparency. The simplest one is to ignore one or more attribute codes at display time. For example, the display routine can assume that the attribute 00H indicates a transparent pixel in the object and skip these pixels at display time.

A more elaborate and sophisticated method of handling transparency is to create a second bitmap that serves as a transparency mask for the original image. Figure 7.11 shows how a transparency mask can be used to determine which pixels in the original bitmap are preserved in the result, and which remain in the background attribute.

The use of a transparency mask can be further refined by specifying different logical operations. For example, a bit field in the mask could determine if the original image is to be logically combined by performing an AND, OR, or XOR operation with mask data contained in another field.

In the bitmap handling procedures listed later in this chapter we have adopted a simple method of transparency handling that is suitable for many applications. It consists of an entry code which sets a transparency switch. If the switch is on, then all 00H pixel codes in the image bitmap are ignored and the background pixel remains unchanged. In this case 00H bitmap codes indicate a transparent attribute. Alternatively, if the transparency switch is off, then pixel codes 00H are displayed according to the current DAC setting for this value.

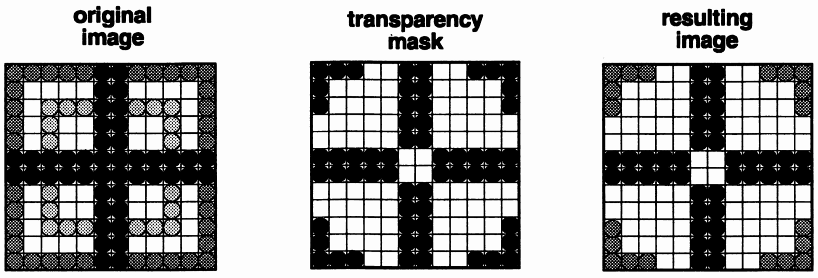


Figure 7.11 *Image Processing by a Transparency Mask*

## 7.5.2 VGA Mode X Bitmap Display

If the 144 pixels in the image of Figure 7.10 are located in four planes, it should be possible to design the output routine so that all pixels located in the same plane are accessed with a single setting of the Map Mask register of the Sequencer. Notice in Figure 7.10 that the pixels in map 0 are located in three columns of the bitmap. If a pointer is set to the start of the bitmap, code can index this pointer to each value that corresponds to a pixel in map 0. At the same time, this routine would have to maintain a destination pointer that keeps track of the video buffer address, as well as several operational counters. The data variables and processing are shown in the following procedure:

```
;****************************************************************
;  code segment variables and constants for bitmap procedures
;                      in VGA mode X
;****************************************************************
H_REPEAT        DB      0    ; Horizontal counter
V_REPEAT        DB      0    ; Vertical counter
PLANE_NUM       DB      0    ; Storage for bitplane
PLANE_CNT       DB      0    ; Bit plane counter
SCREEN_ADD      DW      0    ; Storage for initial screen address
TR_SWITCH       DB      0    ; 0-pixel handling
                             ; 0 = display pixel
                             ; 1 = pixel is transparent
VIDEO_PAGE      DW      0    ; Word storage for video page
;
; Code segment variables for mode X address calculations
EIGHTY      DW      80      ; Pixels per row
FOUR        DB       4      ; Number of planes
THREE_20    DW     320      ; Factor for tile address
;****************************************************************
;                 display bitmap at a pixel level
;****************************************************************
BMAP_OUT_X      PROC    FAR
; Display a bitmap at a pixel-level address while in VGA mode X
; On entry:
;         DS:SI — bitmap
;         BL = number of 4-pixel columns in bitmap
;         BH = number of rows in bitmap
;         CX = x coordinate of pixel address (range 0 to 319)
;         DX = y coordinate of pixel address (range 0 to 239)
;         AL = 0 for 0-pixels displayed (transparency off)
;         AL = 1 for 0-pixels transparent (transparency on)
;         AH = video page number (range 0 to 3)
;              Note: page 3 is a partial page
; Routine logic:
;                Memory bitmap:
;                 R 0 0 0          R = red
;                 0 G 0 0          G = green
;                 0 0 B 0          B = blue
;                0 0 0 W           W = white
;
; Screen:
;           4       5       6       7       8  <==== tile number
;       0 1 2 3|0 1 2 3|0 1 2 3|0 1 2 3|0 1 2 3 <== bit planes
;       R 0 0 0|  R 0 0|0   R 0|0 0   R|0 0 0
;       0 G 0 0|  0 G 0|0   0 G|0 0   0|G 0 0
;       0 0 B 0|  0 0 B|0   0 0|B 0   0|0 B 0
;       0 0 0 W|  0 0 0|W   0 0|0 W   0|0 0 W
;       =======  ========   =======   ========
;         case A   case B     case C    case D_ 3-pixel overlap
;           |         |          |______________ 2-pixel overlap
;           |         |_________________________ 1-pixel overlap
;           |___________________________________ no overlap
; Processing logic:
; 1. Tile counter (CS:PLANE_NUM variable) is initialized to
;    the initial bit plane that results from the display address
;    as follows:
;                bit plane = REM (y/4)
;    For example: in case C the initial bit plane is:
;                REM (6/4) = 2
;
; 2. The bitmap is displayed by planes. If the bitmap is over
;    one tile wide (4 bytes), the routine indexes to successive
;    tiles by adding 4 to the bitmap pointer and incrementing
;    the tile number in the display address (BX)
;
; 3. The bit plane number (CS:PLANE_NUM) is incremented at the er
;    of each iteration. If incrementing the bit plane number
;    causes crossing a tile boundary (PLANE_NUM  3) then:
;                 CS:PLANE_NUM = CS:PLANE_NUM - 4
;    In this case the tile number in the display address
;    (BX) is incremented
;
; 4. Processing ends when 4 bit planes have been displayed
;***************************|
;   save caller's context   |
;***************************|
        PUSH    AX                  ; Save caller's context
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
; Store x and y dimensions of bit plane
        MOV     CS:H_REPEAT,BL      ; Horizontal byte count
        MOV     CS:V_REPEAT,BH      ; Vertical byte count
        MOV     CS:PLANE_CNT,0      ; Initialize plane counter
; Store transparency code
        PUSH    AX                  ; Save page number for later
        MOV     CS:TR_SWITCH,0      ; Assume no transparency
        CMP     AL,1                ; Test transparency entry code
        JNE     STORE_VPAGE     ; Continue if not
        MOV     CS:TR_SWITCH,1  ; Set transparency ON
; Store page number
STORE_VPAGE:
        POP     AX              ; Restore from stack
        MOV     AL,AH           ; Page number to AL
        MOV     AH,0            ; Clear high byte
        MOV     CS:VIDEO_PAGE,AX ; Store it
;***************************|
; calculate buffer address  |
;***************************|
; Formula: offset = (y * 80) + (x/4)
        MOV     AX,DX       ; y coordinate to AX
        MUL     CS:EIGHTY   ; AX = y * 80
        MOV     BX,AX       ; Free AX and hold in BX
        MOV     AX,CX       ; x to AX
; Prepare for division
        DIV     CS:FOUR     ; AX / 4  = quotient in AL and
                            ; remainder in AH
; AH = plane number since plane number = REM (x/4)
; Store bit plane number in variable
        MOV     CS:PLANE_NUM,AH
        MOV     AH,0        ; Clear remainder for addition
        ADD     BX,AX       ; Add into buffer offset
; Calculate and add video page offset
        MOV     AX,19200    ; Length of each page in mode X
        MUL     CS:VIDEO_PAGE ; Multiply by page number
        ADD     BX,AX       ; Add into offset
; At this point BX — video buffer
; Store offset in memory variable
        MOV     CS:SCREEN_ADD,BX
; All bits in the Bit Mask register of the Graphics Controller
; must be set to 1 to allow CPU access
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0FF08H; AL = Bit Mask register
                         ; AH = 11111111B to load maps from
                         ; CPU
        OUT     DX,AX
;***************************|
;    display by bit plane   |
;***************************|
; Display routine indexes to each of the four bit planes in
; mode X by indexing into vertical columns four bytes apart
; Notice that columns 4 bytes apart are in the same bit plane
NEXT_BITPLANE:
        PUSH    SI      ; Save bitmap pointer
; CS:PLANE_NUM holds number of desired bit plane
        MOV     CL,CS:PLANE_NUM ; Plane to CL
; Select active plane by setting the Sequencer's Map Mask register
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0102H ; AL = Map Mask register offset
```

```
                             ; AH = 0000 0001
                             ;              |___ map 0 enabled
; Shift enable bit to left by plane number (in CL)
        SHL     AH,CL        ; If CL = 0 then map 0 enabled
                             ; If CL = x then map x enabled
        OUT     DX,AX
; Reset pixel row counter
        MOV     DH,CS:V_REPEAT    ; Vertical repetitions
;****************************|
;     display bitmap         |
;****************************|
; Bitmap contains entire image
NEXT_PIX_COL:
; BX — offset in video buffer
        MOV     DL,CS:H_REPEAT   ; Reset horizontal counter
        PUSH    BX            ; Save address register
; Display bitmap row
NEXT_PIX_ROW:
        MOV     AL,[SI]   ; Get color code to AL
; Test for 0-value pixel
        CMP     AL,0     ; Is this a 0 pixel code?
        JNE     NOT_0_PIX    ; Go if not
; At this point pixel value = 0
; Check for transparent mode (CS:TR_SWITCH = 1)
        CMP     CS:TR_SWITCH,1    ; Transparent code
        JE      TRANSPARENT_PIX   ; Go if transparency ON
; Code assumes that ES — video buffer segment base (A000H)
NOT_0_PIX:
        MOV     ES:[BX],AL       ; Set pixel
TRANSPARENT_PIX:
        ADD     SI,4             ; Bump bitmap pointer
        INC     BX               ; Bump video pointer
        DEC     DL               ; Decrement counter
        JNZ     NEXT_PIX_ROW
; At this point one row of bitmap is displayed
        POP     BX               ; Address of row start
        ADD     BX,80            ; Index to next row
        DEC     DH               ; Decrement vertical counter
        JNZ     NEXT_PIX_COL
;***************************|
; index to next bit plane   |
;***************************|
; Restore display parameters
        MOV     BX,CS:SCREEN_ADD    ; Video buffer address
        POP     SI                  ; Bitmap pointer
        INC     SI                  ; Bump to next pixel
        INC     CS:PLANE_NUM        ; Next plane
; If plane number is  3 then plane number = plane number - 4
; In this case the display tile must be incremented
        MOV     AL,CS:PLANE_NUM     ; Plane number to AL
        CMP     AL,4                ; Test for limit
        JB      PLANE_IN_RANGE     ; OK if  4
; At this point a plane adjustment is required
        SUB     AL,4               ; Subtract to wrap-around
        MOV     CS:PLANE_NUM,AL    ; and store in variable
        INC     BX                 ; Bump column
        MOV     CS:SCREEN_ADD,BX   ; Update screen address
PLANE_IN_RANGE:
        INC     CS:PLANE_CNT       ; Plane counter
        CMP     CS:PLANE_CNT,4     ; Test for last plane
        JE      EXIT_PLANES        ; Go if last plane
        JMP     NEXT_BITPLANE
EXIT_PLANES:
        POP     SI              ; Restore caller's context
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
BMAP_OUT_X      ENDP
```

### 7.5.3 VGA Mode X Bitmap Acquisition

Graphics applications often need to read and store the contents of a screen area. This area can be as small as a few pixels or as large as the entire video display surface. Animation code often needs to acquire a screen area in order to be able to restore the background image once it is overlaid by the animated object. The sequence of operations usually consists of the following actions:

1. Acquire background.
2. Display animated object.
3. Restore background erasing animated object.

During translation transformations the coordinates of the object position are changed at the conclusion of each cycle of operations.

One alternative is to store the background image in the user's memory space. Later in this chapter we see that sometimes it is convenient to store the background image in video memory in order to further enhance program performance. The acquisition and storage of a screen image in bitmap form can also be done following the plane-by-plane method described in Section 7.5.2. In this case the code must also keep track of two pointers: one to the screen object and another one to the storage area for the acquired image. The indexing of these pointers is based on the architecture of VGA mode X. The following procedure allows acquiring a rectangular screen bitmap and storing it in a buffer designated by the caller.

```
;****************************************************************
;                      read and store bitmap
;****************************************************************
BMAP_IN_X       PROC    FAR
```

```
; Read and store a screen image in bitmap form while in VGA
; mode X
; On entry:
;          DS:DI - storage area for image
;          BL = number of 4-pixel columns in bitmap
;          BH = number of rows in bitmap
;          CX = x coordinate of pixel address (range 0 to 319)
;          DX = y coordinate of pixel address (range 0 to 239)
;          AH = video page number (range 0 to 3)
;               Note: page 3 is a partial page
;
;***************************|
;   save caller's context   |
;***************************|
        PUSH    BX                ; Caller's registers
        PUSH    CX
        PUSH    DX
        PUSH    DI
; Store x and y dimensions of bit plane
        MOV     CS:H_REPEAT,BL    ; Horizontal byte count
        MOV     CS:V_REPEAT,BH    ; Vertical byte count
        MOV     CS:PLANE_CNT,0    ; Initialize plane counter
; Store page number
        MOV     AL,AH             ; Page number to AL
        MOV     AH,0              ; Clear high byte
        MOV     CS:VIDEO_PAGE,AX ; Store it
;
;***************************|
; calculate buffer address  |
;***************************|
; Formula: offset = (y * 80) + (x/4)
GET_BUF:
        MOV     AX,DX         ; y coordinate to AX
        MUL     CS:EIGHTY     ; AX = y * 80
        MOV     BX,AX         ; Free AX and hold in BX
        MOV     AX,CX         ; x to AX
; Prepare for division
        DIV     CS:FOUR       ; AX / 4  = quotient in AL and
                              ; remainder in AH
; AH = plane number since plane number = REM (x/4)
; Store bit plane number in variable
        MOV     CS:PLANE_NUM,AH
        MOV     AH,0          ; Clear remainder for addition
        ADD     BX,AX         ; Add into buffer offset
; Calculate and add video page offset
        MOV     AX,19200      ; Length of each page in mode X
        MUL     CS:VIDEO_PAGE ; Multiply by page number
        ADD     BX,AX         ; Add into offset
; At this point BX -> video buffer
; Store offset in memory variable
        MOV     CS:SCREEN_ADD,BX
```

```
;***************************|
;    read pixel planes      |
;***************************|
; Read routine indexes to each of the 4 bit planes in mode X
; by indexing into vertical columns 4 bytes apart
; Notice that columns 4 bytes apart are in the same bit plane
NEXT_PLANE:
        PUSH    DI      ; and storage pointer
; CS:PLANE_NUM holds number of desired bit plane
        MOV     AH,CS:PLANE_NUM ; Plane to CL
; Select active plane by setting the Graphics Controller Read
; Map Select register
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AL,04H   ; AL = Read Map Select register offset
        OUT     DX,AX
; Reset pixel row counter
        MOV     DH,CS:V_REPEAT   ; Vertical repetitions
;****************************|
;   read and store data      |
;****************************|
NEXT_COL:
; BX — offset in video buffer
        MOV     DL,CS:H_REPEAT  ; Reset horizontal counter
        PUSH    BX          ; Save address register
; Display bitmap row
NEXT_ROW:
        MOV     AL,ES:[BX]      ; Get screen color
        MOV     [DI],AL         ; Store it in memory
        ADD     DI,4            ; and storage pointer
        INC     BX              ; Bump video pointer
        DEC     DL              ; Decrement counter
        JNZ     NEXT_ROW
; At this point one row of bitmap is displayed
        POP     BX              ; Address of row start
        ADD     BX,80           ; Index to next row
        DEC     DH              ; Decrement vertical counter
        JNZ     NEXT_COL
;***************************|
; index to next bit plane   |
;***************************|
; Restore display parameters
        MOV     BX,CS:SCREEN_ADD   ; Video buffer address
        POP     DI                 ; Storage pointer
        INC     DI                 ; Bump to next location
        INC     CS:PLANE_NUM       ; Next plane
; If plane number is  3 then plane number = plane number - 4
; In this case display tile is also must be incremented
        MOV     AL,CS:PLANE_NUM    ; Plane number to AL
        CMP     AL,4               ; Test for limit
        JB      PLANE_OK           ; OK if  4
; At this point a plane adjustment is required
```

```
        SUB     AL,4                ; Subtract to wrap-around
        MOV     CS:PLANE_NUM,AL     ; and store in variable
        INC     BX                  ; Bump column
        MOV     CS:SCREEN_ADD,BX    ; Update screen address
PLANE_OK:
        INC     CS:PLANE_CNT        ; Plane counter
        CMP     CS:PLANE_CNT,4      ; Test for last plane
        JE      EXIT_READ           ; Go if last plane
        JMP     NEXT_PLANE
EXIT_READ:
        POP     DI                ; Restore caller's context
        POP     DX
        POP     CX
        POP     BX
        RET
BMAP_IN_X       ENDP
```

## 7.6 VGA Mode X bitBlt Primitives

In the language of computer graphics a *bitBlt* (pronounced "bit blit") is an operation whereby an entire bitmap is rapidly transferred to another location. In this sense, the procedures to display and to read a bitmap listed in Sections 7.5.2 and 7.5.3 are bitBlts. However, it is more common to designate an operation as a bitBlt when the source and the destination memory are of the same type, for instance, when an image is copied from one area of display memory to another one.

In VGA mode X bitBlt operations can be rapidly executed by means of the parallel pixel manipulation techniques described in Section 7.1 and shown in Figure 7.5. One limitation of the parallel processing of pixel data is that both the source and the destination images must be located at a 4-pixel boundary. However, when this restriction is tolerable, parallel processing of pixel data permits a very fast transfer of the image data from one area of video memory to another one.

### 7.6.1 Page-Level bitBlt

Programs that use multiple screen pages often need to copy an image rectangle from one video page to another one. For example, an animation routine can use this type of bitBlt to save the background context to be overlaid by the animated object. The following procedure allows a video page-to-page bitBlt:

```
;****************************************************************
;  code segment variables and constants for bitBlt procedures
;                         in VGA mode X
;****************************************************************
H_REPEAT        DB      0    ; Horizontal counter
V_REPEAT        DB      0    ; Vertical counter
;
```

```
; Code segment variables for mode X address calculations
EIGHTY      DW      80      ; Pixels per row
; Code segment data for Bitblt procedures
SOURCE_PG       DW      0       ; Source video page
DEST_PG         DW      0       ; Destination video page
;
;****************************************************************
;                          page bitBlt
;****************************************************************
PAGE_BITB_X     PROC    FAR
; Perform a video-to-video memory page Bitblt operation while in
; VGA mode X
; On entry:
;         CX = start x coordinate of source (range 0 to 79)
;         DX = start y coordinate of source (range 0 to 239)
;         BL = x dimension of source (1 to 80)
;         BH = y dimension of source (1 to 240)
;         AL = source video page (range 0 to 2)
;         AH = destination video page (range 0 to 2)
;
;**************************|
;   save caller's context   |
;**************************|
        PUSH    AX              ; Caller's context
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
; Store x and y dimensions of bit plane
        MOV     CS:H_REPEAT,BL  ; Horizontal byte count
        MOV     CS:V_REPEAT,BH  ; Vertical byte count
        PUSH    AX              ; Save pages
; Store page numbers (in word variables)
        MOV     AH,0            ; Clear high byte of word
        MOV     CS:SOURCE_PG,AX ; Store video page of source
        POP     AX              ; Restore pages
        MOV     AL,AH           ; Destination to AL
        MOV     AH,0            ; Clear high byte of word
        MOV     CS:DEST_PG,AX   ; Store video page of destination
;**************************|
; calculate buffer address  |
; of source and destination |
;**************************|
; Formula: offset = (y * 80) + x
        MOV     AX,DX         ; y coordinate to AX
        MUL     CS:EIGHTY     ; AX = y * 80
        MOV     SI,AX         ; Free AX and hold in SI
        ADD     SI,CX         ; Add x
        MOV     DI,SI         ; Copy offset in destination
; Add in video page offset (19200 bytes per page)
; for source page
        MOV     AX,19200    ; Length of video page in mode X
        MUL     CS:SOURCE_PG; Multiply by source page number
        ADD     SI,AX       ; Add page offset to pointer
; At this point SI — source's tile level offset into buffer
        MOV     AX,19200    ; Length of video page in mode X
        MUL     CS:DEST_PG  ; Multiply by destination page number
        ADD     DI,AX       ; Add page offset to pointer
; At this point DI — destination's tile level offset into
; buffer
;**************************|
;  set to read all planes   |
;**************************|
; Select active planes by setting the Graphics Controller Read
; Map Select register
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0F04H ; AL = Read Map Select
                         ; AH = 00001111B to read 4 maps
        OUT     DX,AX
;**************************|
;  set to write all planes  |
;**************************|
; Select active plane by setting the Sequencer's Map Mask
; register
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0F02H ; AL = Map Mask register offset
                         ; AH = 0000 1111
                         ;           ||||___ all maps enabled
        OUT     DX,AX
; All bits in the Bit Mask register of the Graphics Controller
; must be set to 0 to set the destination planes from the latches
; instead of the CPU
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0008H ; AL = Bit Mask register
                         ; AH = 00000000B to load maps from
                         ; latches
        OUT     DX,AX
;***************************|
;   read and write data      |
;***************************|
RW_ROW:
        PUSH    SI                ; Save address pointer of source
        PUSH    DI                ; and destination
        MOV     DL,CS:H_REPEAT    ; Reset horizontal counter
; Display bitmap row
RW_COL:
        MOV     AL,ES:[SI]        ; Read all latched planes
        MOV     ES:[DI],AL        ; Write all latched planes
        INC     SI                ; Bump source pointer
        INC     DI                ; And destination
        DEC     DL                ; Decrement counter
        JNZ     RW_COL
; At this point a row of data has been transferred from the
; source to the destination video page
        POP     DI              ; Restore pointers
        POP     SI
        ADD     SI,80           ; Index to next row
        ADD     DI,80
        DEC     CS:V_REPEAT     ; Decrement vertical counter
        JNZ     RW_ROW
; All rows have been read and stored
        POP     DI              ; Restore caller's context
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
PAGE_BITB_X     ENDP
```

### 7.6.2 Tile-Level BitBlt

In addition to transferring image data from one video page to another one, a graphics application executing in VGA mode X often needs to transfer image data at a smaller-than-page level. In this case, the purpose of the bitBlt routines is to transfer image data in parallel in order to achieve the highest possible performance. However, image data cannot be transferred at the pixel level in VGA mode X without accessing all four bit planes, as is the case in the BMAP_OUT_X and BMAP_IN_X procedures listed in Section 7.5.2 and 7.5.3, respectively. Therefore, there would be no gain in developing a separate bitBlt pixel-level procedure. On the other hand, it is possible to rapidly transfer areas of video memory located at the screen tile level by latching all four maps in a single operation. The following procedure performs the necessary processing:

```
;****************************************************************
;                         tile bitBlt
;****************************************************************
TILE_BITB_X     PROC    FAR
; Perform a video-to-video memory tile bitBlt operation while in
; VGA mode X
;
; On entry:
;        CL = start x coordinate of source (range 0 to 79)
;        CH = start y coordinate of source (range 0 to 239)
;        DL = start x coordinate of destination
;        DH = start y coordinate of destination
;        BL = x dimension of bitmap (1 to 80)
;        BH = y dimension of bitmap (1 to 240)
;        AL = source video page (range 0 to 2)
;        AH = destination video page (range 0 to 2)
```

```
;***************************|
;   save caller's context   |
;***************************|
        PUSH    AX                ; Caller's context
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
; Store x and y dimensions of bit plane
        MOV     CS:H_REPEAT,BL  ; Horizontal byte count
        MOV     CS:V_REPEAT,BH  ; Vertical byte count
        PUSH    AX                ; Save pages
; Store page numbers (in word variables)
        MOV     AH,0              ; Clear high byte of word
        MOV     CS:SOURCE_PG,AX ; Store video page of source
        POP     AX                ; Restore pages
        MOV     AL,AH             ; Destination to AL
        MOV     AH,0              ; Clear high byte of word
        MOV     CS:DEST_PG,AX    ; Store video page of destination
;***************************|
; calculate buffer address  |
;       of source           |
;***************************|
; CL = start x coordinate
; CH = start y coordinate
; Formula: offset = (y * 80) + x
        PUSH    DX                ; Save destination registers
        MOV     AL,CH         ; y coordinate to AX
        MOV     AH,0          ; Clear high byte
        MUL     CS:EIGHTY     ; AX = y * 80
        MOV     SI,AX         ; Free AX and hold in SI
        MOV     CH,0          ; Clear y coordinate
        ADD     SI,CX         ; Add x
; Add in video page offset (19200 bytes per page)
; for source page
        MOV     AX,19200      ; Length of video page in mode X
        MUL     CS:SOURCE_PG; Multiply by source page number
        ADD     SI,AX         ; Add page offset to pointer
; At this point SI — source's tile level offset into buffer
;***************************|
; calculate buffer address  |
;       of destination      |
;***************************|
; DL = start x coordinate
; DH = start y coordinate
; Formula: offset = (y * 80) + x
        POP     DX            ; Restore destination registers
        PUSH    DX            ; Save from word multiply
        MOV     AL,DH         ; y coordinate to AX
        MOV     AH,0          ; Clear high byte
        MUL     CS:EIGHTY   ; AX = y * 80
        MOV     DI,AX       ; Free AX and hold in DI
        POP     DX          ; Restore destination registers
        MOV     DH,0        ; Clear y coordinate
        ADD     DI,DX       ; Add x
; Add in video page offset (19200 bytes per page)
; for destination page
        MOV     AX,19200    ; Length of video page in mode X
        MUL     CS:DEST_PG  ; Multiply by destination page number
        ADD     DI,AX       ; Add page offset to pointer
; At this point DI — destination's tile level offset into buffer
;**************************|
;  set to read all planes   |
;**************************|
; Select active planes by setting the Graphics Controller Read
; Map Select register
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0F04H ; AL = Read Map Select
                         ; AH = 00001111B to read 4 maps
        OUT     DX,AX
;**************************|
;  set to write all planes  |
;**************************|
; Select active plane by setting the Sequencer's Map Mask register
        MOV     DX,3C4H  ; Sequencer base address
        MOV     AX,0F02H ; AL = Map Mask register offset
                         ; AH = 0000 1111
                         ;           ||||___ all maps enabled
        OUT     DX,AX
; All bits in the Bit Mask register of the Graphics Controller
; must be set to 0 to set the destination planes from the latches
; instead of the CPU
        MOV     DX,3CEH  ; Graphics Controller base address
        MOV     AX,0008H ; AL = Bit Mask register
                         ; AH = 00000000B to load maps from
                         ; latches
        OUT     DX,AX
;***************************|
;   read and write data      |
;***************************|
RWT_ROW:
        PUSH    SI              ; Save address pointer of source
        PUSH    DI              ; and destination
        MOV     DL,CS:H_REPEAT  ; Reset horizontal counter
; Display bitmap row
RWT_COL:
        MOV     AL,ES:[SI]      ; Read all latched planes
        MOV     ES:[DI],AL      ; Write all latched planes
        INC     SI              ; Bump source pointer
        INC     DI              ; And destination
        DEC     DL              ; Decrement counter
        JNZ     RWT_COL
; At this point a row of data has been transferred from the
; source to the destination video page
        POP     DI              ; Restore pointers
        POP     SI
        ADD     SI,80           ; Index to next row
        ADD     DI,80
        DEC     CS:V_REPEAT     ; Decrement vertical counter
        JNZ     RWT_ROW
; All rows have been read and stored
        POP     DI              ; Restore caller's context
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
TILE_BITB_X     ENDP
```

## 7.7 Mode X Animation

Animation programming in VGA mode X is subject to the same limitations as in the standard VGA modes. The techniques and manipulations are the same, and the programmer is confronted with similar challenges and difficulties.

One technique, already described, consists of translating a screen object by successively performing a save-display-erase sequence. During the save cycle the screen area to be occupied by the animated object is saved, either in the user's memory space or in an off-screen area of video memory. During the second cycle the animated object is displayed by overlaying it on the background. Finally, the animated object is erased by redisplaying the saved image. In order to smoothly perform this cycle, code must provide some form of timing mechanism. General timing routines are discussed in depth in Part 3 of the book. In this context we casually present several timing methods that can be used in VGA mode X animation programming.

### 7.7.1 Intercepting the Vertical Retrace

A raster-scan display operates by projecting an electron beam on each horizontal row of screen pixels. Pixel scanning proceeds, row by row, from the top left screen corner to the bottom-right. To avoid visible interference, the electron beam is turned off during the period in which the gun is re-aimed back to the start of the next pixel row (horizontal retrace). The beam is also turned off while it is re-aimed from the last pixel on the bottom-right corner of the screen to the first pixel at the top-left corner (vertical retrace). Because of the distance and directions involved, the vertical retrace takes much longer than the horizontal retrace. The screen refresh periods in VGA graphics modes take place at an approximate rate of 70 times per second.

The start of the vertical retrace cycle can be determined by two general methods: method number 1 is reading bit 7 of the VGA Input Status register 0 in the General register group. This bit is set if a vertical retrace is in progress. A second, and more reliable method, is to determine the start of the vertical retrace by reading bit 3 of the Input Status Register 1. In either method, in order to maximize the interference-free time available during a vertical retrace the code must wait for the start of a vertical retrace cycle. This requires first waiting for a vertical retrace cycle to end, if one is in progress, and then detecting the start of a new cycle. The following procedure allows determining the start of the vertical retrace cycle:

```
TIME_VR         PROC        FAR
; Test for start of the vertical retrace cycle of the CRT
; Controller
; Bit 3 of the input status register 0 is set if a vertical cycle
; is in progress
;
; Save caller's context
        PUSH    AX
        PUSH    DX
        MOV     DX,3DAH             ; Input status register 1
                                    ; address in VGA mode X
VRC_CLEAR:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JNZ     VRC_CLEAR           ; Wait until bit clear
                                    ; Vertical retrace ended
VRC_START:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JZ      VRC_START           ; Wait until bit set
                                    ; Vertical retrace has started
        POP     DX                  ; Restore caller's context
        POP     AX
        RET
TIME_VR         ENDP
```

The use of the vertical retrace cycle offers the advantage that the screen update takes place while the video function is off. This reduces interference during animation routines, but only if the screen update takes place before the video function is restored by the VGA. Timing requirements in animation routines are discussed more extensively in Chapter 11.

### 7.7.2 Interval Timing

Animation code often has to provide a wait period in order to ensure that the animated image is retained on the screen or remains erased for a minimum time lapse. This software mechanism is often called an interval timer. The simplest and least satisfactory form of an interval timer is a wait loop. For example, code can produce a delay by means of the following instruction sequence:

```
      MOV       CX,2000          ; Load counter
DELAY_2000:
      AAM                        ; Dummy instruction to delay
                                 ; 17 clock cycles in a 80386
      LOOP      DELAY_2000
```

The main objection to the loop delay method is that the actual time varies according to the system in which the code is executing. For example, the AAM (ASCII Adjust After Multiplication) instruction produces a delay of 17 clock cycles in a 80386 CPU and 83 clock cycles in an 8086 CPU. In addition, the clock speed of the CPU also affects the delay period.

In spite of these hardware differences it is possible to develop routines that dynamically adjust the delay period to the characteristics of the host machine. One approach is to include a delay loop in the initialization code and to time the execution of this loop by the host system. The code can then adjust an internal variable in order make the necessary compensations for a system that is so much slower or faster than a predetermined norm. Perhaps a simpler approach is to use timer channel 0 to provide a system-independent interval timing routine. The following procedure shows the required processing:

```
MILLI_TIME      PROC     FAR
; Timer in 1/1000 second intervals
; On entry:
;          BX = time delay in milliseconds
;
        PUSH    CX                ; Save context
        PUSH    DX
READ_COUNTER:
; Read timer channel 0 at port 40H
        MOV     AL,00000110B  ; 0 0 0 0 0 1 1 0
                              ; ___ ___ _____ _
                              ;  |   |    |   |__ binary mode
                              ;  |   |    |______ mode 3
                              ;  |   |___________ read latch
                              ;  |_______________ channel 0
        OUT     43H,AL        ; To counter command port
; Read LSB then MSB
        IN      AL,40H
        MOV     DL,AL
        IN      AL,40H
        MOV     DH,AL
; DX holds timer counter value
;*******************|
;    count 1 ms     |
;*******************|
        SUB     DX,1190         ; 1 millisecond delay
ONE_MS:
        MOV     AL,06H          ; Latch for read code
        OUT     43H,AL          ; To counter command port
; Read LSB then MSB
        IN      AL,40H          ; Port for channel 0
        MOV     CL,AL           ; LOB to DL
        IN      AL,40H          ; Read HOB
        MOV     CH,AL           ; HOB to DH
; CX holds timer counter value
; DX holds terminal count for 1 millisecond delay
        CMP     CX,DX           ; Compare counts
        JA      ONE_MS          ; Wait until DX  CX
;*******************|
;    repeat BX ms   |
;*******************|
        DEC     BX              ; Milliseconds counter
        JNZ     READ_COUNTER    ; Continue if count not finished
        POP     DX
        POP     CX
        RET
MILLI_TIME      ENDP
```

## 7.7.3 Video Paging in VGA Mode X

The availability of several screen pages in VGA mode X (see Figure 7.9) provides a method that allows translation animation by storing different images in the various screen pages. In this case code can follow the save-display-erase cycle by manipulating screen pages simultaneously. One possible method consists of devoting a screen page to storing the image background, excluding the animated object. The code manipulates the background and the object image in a second screen page. The display cycle consists of alternating the background page (object erased) and the image and background page (object displayed). The object updates are performed while the object and image page are off screen, thus increasing code performance and eliminating interference.

In order to produce paging animation in VGA mode X a routine that allows selecting the active video page is required. The following procedure performs this function:

```
PAGE_ON_X       PROC    FAR
; Select display page while in VGA mode X
; On entry:
;         AL = video page to display (range 0 to 2)
;
        MOV     CL,AL           ; Page number to CL
        MOV     CH,0            ; Clear high byte
        MOV     AX,19200        ; Select start of buffer
        MUL     CX              ; AX * page number
        MOV     BX,AX           ; Buffer offset to BX
        MOV     DX,03D4H        ; CRT Controller base address
                                ; in VGA mode X
        CALL    FAR PTR TIME_VR ; Wait for vertical retrace cycle
        MOV     AL,0CH     ; Offset of Start Address High
                           ; register
        MOV     AH,BH      ; High byte of video buffer address
        OUT     DX,AX      ; To Start Address High register
        INC     AL         ; Offset of Start Address Low register
        MOV     AH,BL      ; Low byte of video buffer address
        OUT     DX,AX      ; To Start Address Low register
        RET
PAGE_ON_X       ENDP
```

### 7.7.4 VGA Mode X Panning Animation

Panning animation in some VGA modes, including mode X, can be performed by changing the address of the video buffer that is mapped to the display function. The start address is stored in the Start Address High and Start Address Low registers of the CRT Controller group. On VGA initialization these registers are set so that the video function is mapped to offset 0 in the buffer; that is, the object displayed at the top-left screen position is stored as the start of the video segment located at physical address A0000H.

Code can scroll up the screen by increasing the start address by one pixel row, which corresponds to an increment of 80 bytes in VGA mode X. A faster horizontal panning can be produced by incrementing the start address in multiples of 80 bytes. Slow panning on the vertical plane is produced by incrementing the buffer start address by one. The following procedures provide means for panning horizontally and vertically in VGA mode X:

```
PAN_RIGHT_X     PROC    NEAR
; Pan screen horizontally, left-to-right, in VGA mode X
; On entry:
;       CX = number of screen columns to pan (1 to 79)
;       BX = start address in video buffer
;
; Prepare to pan screen
; BX holds the start address of the video buffer. BH holds
; the high byte and BL the low byte
        MOV     DX,03D4H        ; CRT Controller base address
                                ; in VGA mode X
PAN_RIGHT_ONE:
        CALL    FAR PTR TIME_VR ; Wait for vertical retrace cycle
        MOV     AL,0CH     ; Offset of Start Address High
                           ; register
        MOV     AH,BH      ; High byte of video buffer address
        OUT     DX,AX      ; To Start Address High register
        INC     AL         ; Offset of Start Address Low register
        MOV     AH,BL      ; Low byte of video buffer address
        OUT     DX,AX      ; To Start Address Low register
        SUB     BX,1        ; Index to next column
        LOOP    PAN_RIGHT_ONE
        RET
PAN_RIGHT_X     ENDP
;****************************************************************
;
PAN_LEFT_X      PROC    FAR
; Pan screen horizontally, right-to-left, in VGA mode X
; On entry:
;         CX = number of screen columns to pan (1 to 79)
;         BX = start address in video buffer
;
; Prepare to pan screen
; BX will hold the start address of the video buffer. BH holds
; the high byte and BL the low byte
        MOV     DX,03D4H         ; CRT Controller base address
                                 ; in VGA mode X
PAN_LEFT_ONE:
        CALL    FAR PTR TIME_VR ; Wait for vertical retrace cycle
        MOV     AL,0CH      ; Offset of Start Address High
                            ; register
        MOV     AH,BH       ; High byte of video buffer address
        OUT     DX,AX       ; To Start Address High register
        INC     AL          ; Offset of Start Address Low register
        MOV     AH,BL       ; Low byte of video buffer address
        OUT     DX,AX       ; To Start Address Low register
        ADD     BX,1        ; Index to next column
        LOOP    PAN_LEFT_ONE
        RET
PAN_LEFT_X      ENDP
;****************************************************************
;
PAN_UP_X        PROC    FAR
; Pan up 2 screen rows in VGA mode X
; On entry:
;         CX = number of screen columns to pan (1 to 239)
;         BX = start address in video buffer
;
; Prepare to pan screen
; BX holds the start address of the video buffer. BH holds
; the high byte and BL the low byte
        MOV     DX,03D4H         ; CRT Controller base address
                                 ; in VGA mode X
PAN_UP_ONE:
        CALL    FAR PTR TIME_VR ; Wait for vertical retrace cycle
        MOV     AL,0CH      ; Offset of Start Address High
                            ; register
        MOV     AH,BH       ; High byte of video buffer address
        OUT     DX,AX       ; To Start Address High register
        INC     AL          ; Offset of Start Address Low register
        MOV     AH,BL       ; Low byte of video buffer address
        OUT     DX,AX       ; To Start Address Low register
        ADD     BX,160      ; Index to next column
        LOOP    PAN_UP_ONE
        RET
PAN_UP_X        ENDP
;****************************************************************
;
PAN_DOWN_X      PROC    FAR
; Pan down 2 screen rows in VGA mode X
; On entry:
;         CX = number of screen columns to pan (1 to 239)
;         BX = start address in video buffer
;
; Prepare to pan screen
; BX holds the start address of the video buffer. BH holds
; the high byte and BL the low byte
        MOV     DX,03D4H        ; CRT Controller base address
                                ; in VGA mode X
PAN_DW_ONE:
        CALL    FAR PTR TIME_VR ; Wait for vertical retrace cycle
        MOV     AL,0CH      ; Offset of Start Address High
                            ; register
        MOV     AH,BH       ; High byte of video buffer address
        OUT     DX,AX       ; To Start Address High register
        INC     AL          ; Offset of Start Address Low register
        MOV     AH,BL       ; Low byte of video buffer address
        OUT     DX,AX       ; To Start Address Low register
        SUB     BX,160      ; Index to next column
        LOOP    PAN_DW_ONE
        RET
PAN_DOWN_X      ENDP
```

Chapter

8

# XGA Architecture and Initialization

## 8.0 The IBM Extended Graphics Array Video System (XGA)

In September 1990 IBM disclosed preliminary information on a new graphics standard designated as the Extended Graphics Array, or XGA. Soon thereafter two configurations of the XGA were made available: as an adapter card and as part of the motherboard. The XGA adapter is compatible with PS/2 Micro Channel machines equipped with the 80386, 80386SX, and 486 CPU. Therefore it cannot be used in the Model 50, 60, and other machines equipped with the 80286 CPU. The XGA system is integrated in the motherboard of IBM Model 95 XP 486. XGA was also developed at the IBM UK Labs Ltd.

IBM has furnished XGA hardware information and entered into agreements with Intel and other companies for the development of XGA-based products. Several XGA cards are available for non-IBM compatible PCs.

The following are the most notable features of the XGA video system:

1. The maximum resolution is 1024-by-768 pixels in 256 colors.
2. The XGA system is compatible with the 8514/A Adapter Interface software.
3. The display driver for the original version of XGA is interlaced at 1024-by-768 pixel resolution. In 1992 IBM introduced a noninterlaced version of XGA, designated as XGA-2 or XGA-NI.
4. The adapter version of XGA is furnished with either 512K or 1Mb of on-board video RAM. However, the XGA-2 upgrade is always equipped with 1Mb of VRAM.
5. The XGA is compatible with the VGA standard at the register level. This makes possible the use of XGA in the motherboard while still maintaining VGA functionality, as in IBM Model 95 XP 486 microcomputer.
6. XGA includes two original display modes that are not available in its predecessor system 8514/A: a 132-column text mode and a direct color graphics mode with 640-by-480 pixel resolution in 64K colors. This graphics mode is available only in cards with 1Mb video RAM installed.
7. XGA requires a machine equipped with a 80386, 486, or Pentium CPU.

8. XGA implements a three-dimension, user-definable drawing space, called a bitmap. XGA bitmaps can reside anywhere in the system's memory space. The application can define a bitmap in the program's data space and the XGA uses this area directly for drawing, reading, and writing operations.
9. XGA is equipped with a hardware-controlled graphics cursor, called the *sprite*. It maximum size is 64-by-64 pixels and it can be positioned anywhere on the screen without affecting the image stored in video memory.
10. The XGA Adapter Interface is implemented as a .SYS device driver. The module name for the XGA driver is XGAAIDOS.SYS.
11. The XGA was designed taking into consideration the problems associated with managing a video image in a multitasking environment. Therefore it contains facilities for saving and restoring the state of the video hardware.
12. The XGA hardware can act as a bus master and access system memory directly. This bus-mastering capability frees the CPU for other tasks while the XGA graphics coprocessor is accessing memory.
13. IBM has provided register-level documentation for the XGA system. This information allows hardware programming of the XGA. In fact, IBM seems to favor register programming over the AI interface. The hardware data has also facilitated cloning of the XGA system.

Some criticism has been raised regarding XGA, for example, the micro channel requirement in the IBM version of XGA and the limitations of the AI services. The objection to interlaced display technology applies only to the original version of XGA, since the XGA-2 version is noninterlaced. Figure 8.1 is a diagram of the XGA system.

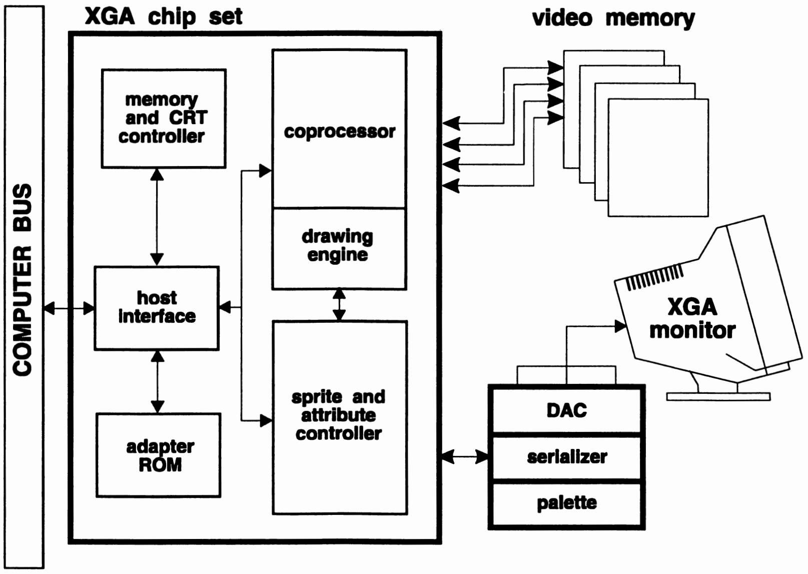


Figure 8.1 *XGA Video System Diagram*

### 8.0.1 Technical Description

The IBM XGA Display Adapter/A is designed for micro channel computers equipped with the 80386, 30386SX, 486, or Pentium CPU. The exception is the IBM Model 70 Portable, which does not accept the XGA adapter. Non-IBM versions of XGA can be used in machines based on the ISA and EISA standards. The IBM XGA Display Adapter can be installed in any 16-bit or 32-bit slot, except the Auxiliary Video Extension slot. XGA is furnished on the motherboard of some high-end models of the PS/2 line. XGA includes all VGA modes and is hardware compatible with the VGA standard. XGA is supplied with device drivers for Windows, OS/2, and some high-end graphics applications.

A maximum of eight XGA Display Adapters can coexist in one system unit. In systems in which XGA is furnished on the motherboard the maximum is five. In IBM documentation each occurrence of an XGA Display Adapter/A is called an "instance." Various instances of XGA are located at different memory addresses. This allows the control of each instance independently. Since VGA has no support for multiple instances, only one video subsystem can be active in a VGA mode. The XGA Display Adapter/A is equipped with software in ROM to perform a system self-test and to initialize the host interface.

### 8.0.2 XGA in VGA Modes

In systems in which XGA is furnished in the motherboard, it is the default VGA (assuming that a display is attached to the connector). If no display is attached, the XGA Adapter with a display connected, installed in the first slot, is the default VGA. In this case all other XGA Display Adapters in the system are configured into Extended Graphics mode.

In systems in which XGA is furnished as a display adapter, the one in the Base Video extension slot is the default VGA, if a display is attached. If no display is attached, or an XGA Display Adapter/A is not installed in the Base Video extension, then the first XGA Display Adapter in the system with a display attached is the default VGA. In this case all other XGA Display Adapters in the system unit are configured in Extended Graphics mode.

### 8.0.3 Multiple XGA Systems

The VGA system, or the VGA function in an XGA system, uses fixed addresses and port designations. For this reason a single VGA or VGA-capable system is active at one time. However, an application running in a machine with more than one VGA or VGA-capable system can select which one is currently enabled. Since a disabled VGA continues to display screen data, the programming can be designed so that the user perceives the system as a multiple-screen setup. A program can enable or disable a particular VGA system by accessing the XGA Operating Mode register of the Display Controller.

A machine with sufficient expansion slots can accommodate up to eight XGA subsystems. Since each instance of the XGA uses different port and address designations, all of them can operate simultaneously in non-VGA modes. In this case the application can access each XGA instance without concern for the others being enabled or disabled.

## 8.0.4 XGA Extended Graphics Modes

As in some VGA modes, XGA video memory is organized in bit planes. Each bit plane encodes the color for a rectangular array of 1024-by-1024 pixels. In practice, since the highest available resolution is 1024-by-768 pixels, there are 256 unused bits in each plane. This unassigned area is used by AI software as a scratchpad during area fills and in marker manipulations, as well as for storing bitmaps for the character sets.

XGA operates in one of two modes: low-resolution and high-resolution. The characteristics of these modes are shown in Table 8.1.

Table 8.1 *XGA Advanced Function Modes*

| | LOW-RESOLUTION MODE | HIGH-RESOLUTION MODE |
|---|---|---|
| RAM installed | 512K | 1024K |
| Interlaced | NO | YES |
| Pixel columns | 640 | 1024 |
| Pixel rows | 480 | 768 |
| Number of colors | 16 | 256 |
| Palette | 256K | 256K |

When the graphics system is in the low-resolution mode, video memory consists of eight 1024-by-512 bit planes. However, the eight bit planes are divided into two separate groups of four bit planes each. These two bit plane groups can be simultaneously addressed. In low-resolution mode the color range is limited to 16 simultaneous colors. In the high-resolution mode (see Table 8.1) video memory consists of eight bit planes of 1024-by-1024 pixels and the number of simultaneous colors is 256. Figure 8.2 shows the bit-plane mapping in XGA high-resolution modes.

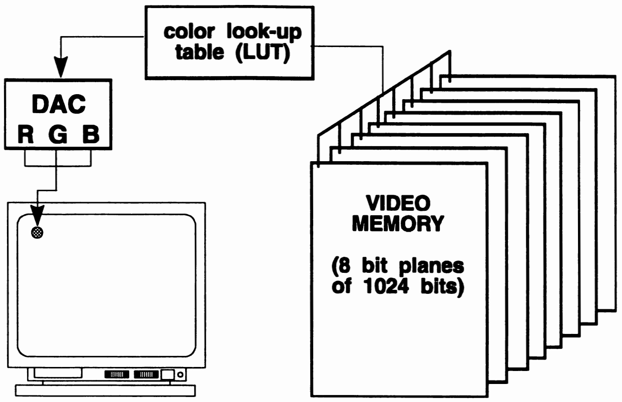


Figure 8.2 *Architecture of the XGA High-Resolution Modes*

Color selection is performed by means of a color *look-up table* (LUT) associated with the DAC. The selection mechanism is similar to the one used in VGA mode number 19; in other words, the 8-bit color code stored in XGA video memory serves as an index into the color look-up table.

### 8.0.5 Alphanumeric Support

The XGA Adapter Interface provides services for the display of text strings and of individual characters. The string-oriented services are designated as *text functions* in the AI documentation, while the character-oriented services are called *alphanumeric functions*. The AI text and character display services are necessary since BIOS and DOS functions for displaying text do not operate on the XGA.

Both text and alphanumeric functions in the AI require the use of character fonts, furnished in the XGA software package. These character fonts are stored in disk files located in the adapter's support diskette. During installation the font files are moved to a specially designated directory in the user's hard disk drive. There are four standard fonts in the XGA diskette. In addition, the XGA diskette contains four supplementary fonts that have been optimized for the XGA hardware. The diskette furnished with IBM Personal System/2 Display Adapter 8514/A Adapter Interface Programmer's Guide contains 22 additional fonts, which are also compatible with the XGA system.

### 8.0.6 The Adapter Interface

The Adapter Interface (AI) is a software package furnished with 8514/A and XGA systems that provides a series of low-level services to the graphics programmer. In the 8514/A the AI software is in the form of a terminate and stay resident program, while in the XGA it is supplied as a SYS-type driver. Since the functions provided by the AI are limited and execution using the interface is slower than directly programming the XGA hardware, the AI is not discussed further in this book.

### 8.0.7 Multidisplay Graphics Systems

The possibility of multidisplay XGA systems creates a new potential for PC graphics applications and system programs. For example, by manipulating the XGA address decoding mechanism an application can display different data on multiple XGA screens. In this manner it is possible to conceive an XGA multitasking environment that manipulates several displays. One feasible setup for a multidisplay system would be an airline scheduling software package that shows and updates arrivals on one screen and departures on another one, while a third monitor is actively attached to the reservations desk. In a graphics applications environment we can envision a desktop publishing system in which the central monitor displays the typesetting software, the monitor on one side is attached to a graphics illustration program, and the one on the other side to a text editor. The user can shift between programs while the inactive software continues processing as a background task. In conclusion, multidisplay, multitasking graphics software could considerably expand the horizons of microcomputing.

## 8.1 XGA Architecture

The XGA system, whether it be furnished on the motherboard or as a plug-in adapter, has the following components:

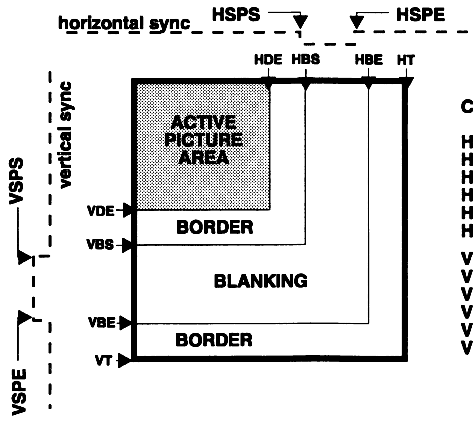
1 A CRT Controller
2. A video buffer
3. A Serializer/Palette/Digital-Analog Converter (SPD)
4. A Sprite/Attribute Controller

### 8.1.1 XGA CRT Controller

The XGA CRT Controller, also called the Display Controller, includes a host interface, a CRT Controller, and a graphics coprocessor. The host interface consists of a group of registers, a pixel interface mechanism, a Bus Master Controller, and a module for saving and restoring the XGA state. The CPU accesses the XGA functions by means of the host interface registers. Access to video and to system memory is by the Bus Master and the Video RAM Controller. The save-restore module provides task switching support in a multitasking environment, such as in the Windows and OS/2 operating systems.

The host interface determines if the XGA adapter is in a 16- or 32-bit slot and sets up for data transfer operations accordingly. The interface also handles address and I/O register decoding and access to the coprocessor's memory-mapped registers and to the video buffer. When the video subsystem is in its bus master mode, the host interface generates the necessary signals for transferring data between the video buffer and system memory. Note that, once initialized, this data transfer is performed without intervention of the CPU.

The CRT Controller provides the VGA functions. It also generates the timing and control signals necessary to drive the serializer and the DAC. Figure 8.3 is a graphics representation of the functions of the CRT Controller registers.



**CRT Controller registers:**

HT = Horizontal Total
HDE = Horizontal Display End
HBS = Horizontal Blanking Start
HBE = Horizontal Blanking End
HSPS = Horizontal Sync Pulse Start
HSPE = Horizontal Sync Pulse End

VT = Vertical Total
VDE = Vertical Display End
VBS = Vertical Blanking Start
VBE = Vertical Blanking End
VSPS = Vertical Sync Pulse Start
VSPE = Vertical Sync Pulse End

Figure 8.3 *XGA CRT Controller Registers*

The XGA system can be programmed so that an interrupt takes place at the start of the refresh cycle of the CRT Controller. This is accomplished by setting a bit in the Interrupt Enable register. A matching bit in the Interrupt Status register reflects the state of the refresh interrupt. This manipulation is useful in providing a pulse for animation routines with automatic assurance that the screen update functions are performed during the Controller's refresh cycle.

The coprocessor enhances system performance by means of the following functions:

1. Provides drawing functions in hardware. In performing these operations the XGA coprocessor can access both video and system memory.
2. Updates video and system memory independently of the CPU. This allows the main processor and the XGA graphics coprocessor to execute in parallel.
3. Serves as a bus master by directly accessing system memory. In this function the coprocessor follows the general bus-mastering rules regarding arbitration and fairness levels. The coprocessor can act as a bus master for other devices and performs burst mode data transfers at a rate of up to 16Mb per second.
4. Supports virtual memory addressing and provides the rapid suspension and resumption of tasks in a multitasking environment.

### 8.1.2 XGA Video Buffer

The video buffer is the XGA memory area devoted to storing video data. The video buffer is also called video memory, video RAM, and VRAM. The XGA video buffer is dual-ported; that is, it can be accessed by the XGA hardware and by the CPU. The original version of the XGA adapter was furnished with either 512K or 1Mb of VRAM. Table 8.2 shows the relationship between resolution and color range in XGA systems equipped with 512K or 1Mb of VRAM.

Table 8.2 *XGA Resolution and Color*

| VRAM | RESOLUTION | COLORS | |
|---|---|---|---|
| | | DISPLAYED | RANGE |
| 512Kb | 640-by-480 | 256 | 256K |
| | 1024-by-768 | 16 | 256K |
| 1Mb | 640-by-480 | 65,536 | ---- |
| | 1024-by-768 | 256 | 256K |

XGA systems equipped with 1Mb of VRAM perform faster than those equipped with 512Kb due to the fact that the data path is 32 bits wide in 1Mb systems, while it is only 16 bits wide in 512K systems. Note that the XGA-2 version of the adapter is always furnished with 1Mb of VRAM.

### 8.1.3 The Serializer/Palette/Digital-Analog Converter

The *Serializer / Palette / Digital-Analog* converter chip is referred to as the SPD in IBM documentation.

In the original XGA adapters the digital-to-analog converter (DAC) receives a color value encoded into three 6-bit fields and converts it into analog output to the monitor. This operation is consistent with the one performed by the VGA DAC. In the XGA-2 version of the adapter the DAC has been redesigned to support 8 bits per color. Therefore the range was expanded from 262,144 to 16,777,216 possible colors.

The serializer converts the picture data stored in the video buffer into a serial bit stream. The width of the bit stream matches the number of bits per pixel in the current display mode. The valid range is 1, 2, 4, 8, or 16 bits per pixel. The 16-bit-per-pixel mode is also called the direct color mode. In this case the palette registers are bypassed and the 16-bit code is treated by the DAC as red-green-blue encoding in the format shown in Figure 8.4.

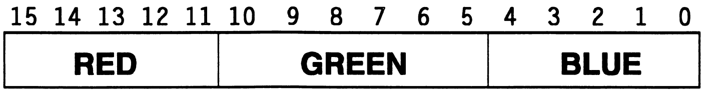


Figure 8.4 *XGA Color Maps in Direct Color Modes*

The palette is a set of registers that controls the actual colors displayed in the 1-, 2-, 4-, and 8-bits-per-pixel modes. The action performed by the palette registers has been compared to that of an artist selecting and mixing the colors to be used in a particular painting. For example, to paint a sunset scene an artist may select shades of red, orange, and yellow, while not using green and blue. By the same token, to represent this scene in a 256-color mode, the programmer would load the palette registers so as to enhance the red, orange, and yellow range, while omitting those colors not used in the scene to be displayed. The limitation of palette manipulation methods is that only one palette can be active for the entire screen. Therefore images that require different palette settings cannot be displayed simultaneously.

## 8.1.4 The XGA Sprite/Attribute Controller

One of the features introduced by the XGA standard, which has no precedence in VGA, is a small screen image, called the *sprite*, controlled separately by the hardware. The sprite is a rectangular screen area of a maximum size of 64-by-64 pixels, which is visible in the XGA graphics modes. On a standard monitor, at a resolution of 1024-by-768, the largest possible sprite image occupies slightly less than one square inch of screen space.

Often a graphics program, at either the system or the application level, must manipulate some sort of animated screen marker object. A typical example of screen marker is a mouse-controlled icon often used to select from option boxes or menus. In PC video systems preceding the XGA the software had to find ways for saving and restoring the screen contents as the maker was moved over the pixel grid. This matter was discussed in Chapter 7 regarding animation in VGA

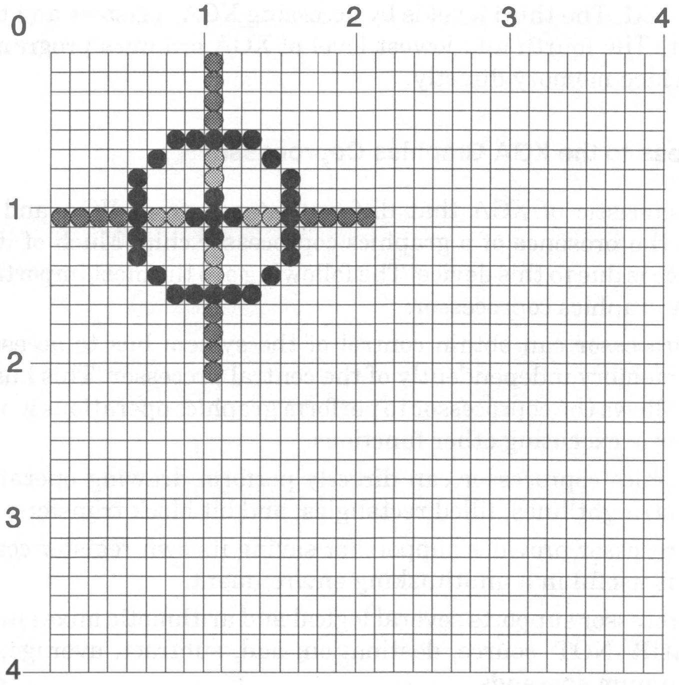


Figure 8.5 *Image Stored in the XGA Sprite Buffer*

mode X. In XGA, the operation of a screen icon is considerably simplified due to the fact that the sprite overlays other screen data without changing it.

The sprite image is stored in a 32K static RAM chip, not part of video memory, usually called the alpha/sprite buffer. This buffer is used for storing alphanumeric characters when XGA is in a VGA mode or in its proprietary 132-column text mode. In these modes the sprite is not visible. Figure 8.5 shows a marker image in the form of a cross-hair symbol located in the sprite buffer.

Note that the marker image in Figure 8.5 is smaller than the 64-by-64 pixels of the sprite buffer. If this is the case, the software can control which part of the sprite image is displayed by manipulating the sprite registers located in the Indexed Access registers of the Display Controller group. In VGA modes the Sprite/Attribute Controller handles color selection and character generation.

## 8.2 Access and Control of the XGA System

The XGA graphics system can be accessed at four different levels. The first and highest level accesses are the graphics functions offered by operating systems and graphics environments. Such is the case in applications that execute under the Windows and OS/2 operating systems and use the graphics services provided by the system software. The second level of XGA programming is by means of a software package furnished with the system, called the *Adapter*

*Interface*, or AI. The third level is by accessing XGA registers and the graphics coprocessor. The fourth and lowest level of XGA graphics programming is by accessing video memory directly.

## 8.2.1 Access to the XGA Graphics Coprocessor

One characteristic of XGA that differentiates it from VGA and SuperVGA systems is the presence of a graphics coprocessor chip. Much of its enhanced performance is due to this device. The following are the most important features of the XGA graphics coprocessor:

1. The coprocessor can obtain control of the system bus to access video and system memory independently of the central processor. This bus-mastering feature allows the coprocessor to perform graphics operations while the main processor is executing other functions.
2. The graphics coprocessor can directly perform drawing operations. These include straight lines, filled rectangles, and bit block transfers.
3. The coprocessor provides support for saving its own register contents. This feature is useful in a multitasking environment.
4. The coprocessor supports several logical and arithmetic mixes including OR, AND, XOR, NOT, source, destination, add, subtract, average, maximum, and minimum operands.
5. The coprocessor can manipulate images encoded in 1-, 2-, 4-, or 8-bits per pixel formats. Pixel maps can be defined in either Intel or Motorola data storage formats.
6. The coprocessor can be programmed to generate system interrupts. These interrupts can occur when the coprocessor operation has finished, an access to the coprocessor was rejected, a sprite operation was completed, or at the end or start of the screen blanking cycle.

The coprocessor registers are memory-mapped. To an application, programming the coprocessor consists of reading and storing data into these reserved memory addresses. In contrast, the XGA main registers are port-mapped and programming consists of reading and writing to these dedicated ports.

The execution of a coprocessor operation consists of the following steps:

1. The system microprocessor reads and writes data to the coprocessor registers that must be initialized for the particular operation.
2. The coprocessor operation starts when a command is written to its Pixel Operations register.
3. The coprocessor executes the programmed operation. During this time the system microprocessor can be performing other tasks. The only possible interference between processor and coprocessor is when both are accessing the bus simultaneously. In this case the access takes place according to established priorities.
4. At the conclusion of the programmed operation the graphics coprocessor informs the system and becomes idle.

## 8.3 XGA Video Memory

Like all PC video systems, XGA is memory-mapped. Therefore the color code for each screen pixel is encoded and stored in video RAM. The number of memory units used to encode the pixel's color depends on the adopted format. Possible values are of 1, 2, 4, 8, and 16 bits per pixel. The number of colors are the respective powers of 2, as shown in Table 8.3.

Table 8.3 *XGA Pixel-to-Memory Mapping*

| BITS PER PIXEL | POWER OF 2 | NUMBER OF COLORS |
|---|---|---|
| 1 | $2^1$ | 2 |
| 2 | $2^2$ | 4 |
| 4 | $2^4$ | 16 |
| 8 | $2^8$ | 256 |
| 16 | $2^{16}$ | 65536 |

The 256- and 65536-color modes are available only in XGA systems with maximum on-board RAM (1Mb). The total amount of VRAM used in storing the image data depends on the resolution and the number of encoded colors. For example, to store the contents of the entire XGA screen at 1024-by-768 pixels resolution in 256 colors requires a total of 786,432 bytes. However, this same screen can be stored in 98,304 bytes if each screen pixel is represented in only two colors and encoded in a single memory bit.

### 8.3.1 Video Memory Apertures

An XGA system can access video memory by means of three different apertures:

1. The first and largest memory aperture is represented in a 22-bit field, which can address 4,194,303 bytes. This 4Mb range allows access to four times the maximum VRAM that can be present in an XGA system, indicating the possibility of a future expansion of the XGA standard. The 4Mb address space requires the use of a 80386 or 486 extended register. This is the aperture used by the XGA graphics coprocessor. The 4Mb aperture is not available in systems in which the XGA is installed in a 16-bit slot or which use the 80386SX CPU.
2. The second possible aperture into video memory is 1Mb, which is also the maximum VRAM that can be present in an XGA system. The 1Mb aperture allows addressing all video memory consecutively by means of an 80386 or 486 extended register.
3. The third possible aperture is also 1Mb, but the total memory space is divided into 16 banks of 64K each. This aperture requires bank switching to access the maximum VRAM. Note that in a particular display mode not all 16 banks could be required to access the mapped video memory space.

By reading the contents of the PS/2 *Programmable Option Select* registers (POS), an application determines which aperture is available in a particular system. The programming operations for detecting and initializing the XGA are described in Chapter 6.

### 8.3.2 Data Ordering

XGA memory mapping can be according to the Intel or the Motorola storage conventions. The XGA hardware allows selecting the Intel or Motorola formats for every operation that accesses a pixel map or image stored in system or video memory. In the Intel convention, also known as the *little-endian* addressing scheme, the smallest element (little end) of a number is stored at the lowest numbered memory location. In the Motorola convention, known as *big-endian* addressing, the largest element (big end) is stored at the lowest numbered memory location. In both the Intel and Motorola storage schemes, the value of individual bits within the stored byte is the same; that is, the low-order bit (bit number 0) is always located at the right-most position.

## 8.4 XGA Detection and Initialization

XGA system detection or initialization functions can be performed by the following methods:

1. By accessing and programming the XGA hardware directly.
2. By means of the *Display Mode Query and Set* function (DMQS) introduced in the XGA-2 standard.
3. By means of VESA XGA services.
4. By the initialization command in the AI software.

Notice that not all methods of XGA detection and initialization offer the same information or produce the same results, also that the various methods are not exclusive. For example, software may use the DMQS function to obtain data about the available options and then proceed to set the XGA graphics mode by accessing the hardware directly. By the same token, a program can use a VESA XGA BIOS service to detect the installed hardware and then use an AI command to initialize the XGA system and set the desired mode.

Software must also take into consideration that not all detection and initialization resources are available in all XGA systems. For example, the XGA AI is a programming convenience which may not be available in a particular implementation. The same applies to the VESA XGA BIOS services. Furthermore, the DMQS function does not exist in XGA systems manufactured before the XGA-2 standard. In fact, the only initialization methods that work in any XGA, regardless of version and options, are those that access the hardware directly. Because of this general usefulness and portability, in the following sections we concentrate our attention on the low-level methods of XGA detection and initialization.

### 8.4.1 Programming the XGA Display Controller

The main programmable device of the XGA system is the Display Controller chip. This IC contains the color look-up table, the CRT Controller, and the hardware registers for the operation of a special cursor, called the sprite. The XGA Display Controller registers are a superset of the VGA registers. As in the VGA, these registers are mapped into the system's I/O space. Therefore they are accessed by the software through input and output ports.

The base address of the XGA Display Controller is usually expressed in the form:

```
21x0H
```

where the variable $x$ depends on the *instance* of the XGA adapter. Recalling that more than one XGA system can coexist in a microcomputer, the instance can be defined as the number that corresponds to a particular XGA adapter or motherboard implementation. In micro channel systems the user can change the instance of an installed XGA adapter by means of the setup procedures provided by the reference diskette. The default instance for a single XGA adapter card is 6, which determines a base address for the Display Controller of 2160H. Note that the instance number replaces the variable $x$ in the general formula.

The XGA standard establishes video modes that are reminiscent of the ones in the VGA system. Table 8.4 lists the characteristics of the various XGA video modes.

Table 8.4 *XGA Video Modes*

| MODE NUMBER | TYPE | HORIZONTAL PIXELS | VERTICAL PIXELS | COLORS |
|---|---|---|---|---|
| 1 | 132-column text | | | |
| 2 | graphics | 1024 | 768 | 256 |
| 3 | graphics | 1024 | 768 | 16 |
| 4 | graphics | 640 | 480 | 256 |
| 5 | direct color | 640 | 480 | 65536 |

The Display Controller registers are divided into two groups: direct access and Indexed Access registers. The Direct Access registers are 10 registers in the range 21x0H to 21x9H. The Indexed Access registers are associated with the Index register (port 21xAH) and the Data registers (ports 21xBH to 21xFH). The Index register values are in the range 04H to 70H but not all values in this range are actually used in XGA. The Direct Access registers in the Display Controller are programmed by means of IN or OUT instructions to the corresponding port; for example, the Memory Access Mode register, at 21x9H can be programmed for eight bits per pixel and Intel data format as follows:

```
MOV     DX,XGA_REG_BASE ; Register base
ADD     DX,9            ; Add offset of Memory Access
                        ; Mode register
```

```
        MOV     AL,00000011B    ; Bitmap for Intel format
                                ; and 8 bits per pixel
        OUT     DX,AL
```

This code fragment assumes that the base address of the Display Controller register groups has been previously determined and is stored in the variable XGA_REG_BASE. The operations necessary for obtaining the base address are discussed later in the chapter. Programming the indexed-access registers takes place in two steps: first, the desired register is selected by writing a value to the Index register at port 21xAH; and second, data is read or written to the register by means of the Data registers in the range 21xBH to 21xFH. The following fragment shows writing all 1 bits (FFH) to the Palette Mask register at offset 64H of the Index register.

```
; Programming an Indexed Access register of the XGA Display
; Controller group
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; Add offset of Index register
        MOV     AL,64H          ; Select Palette Mask register
                                ; at offset 64H
        MOV     AH,0FFH         ; Data byte to write
        OUT     DX,AX           ; Select and write data
```

Notice that the 80x86 instruction OUT DX,AX writes the value in AL to the port number in DX and the value in AH to the port number in DX + 1. Therefore, by using this form of the OUT instruction we can select and access the Index register with a single operation.

## 8.4.2 XGA Hardware Initialization

The first programming operation usually consists of initializing and enabling the XGA video system hardware and software. The simplest initialization method is by means of the AI services previously mentioned. However, device drivers and applications that access the XGA hardware usually initialize the system directly.

For each XGA system the following initialization and diagnostic operations are performed:

1. Locate the XGA hardware and determine if implementation is an adapter or or is the motherboard.
2. Determine the instance and use this value to calculate the XGA register base address.
3. Determine the number of monitors in use. One possibility is that a single monitor provides XGA and VGA functions. Another possibility is a dual-monitor setup, one for XGA modes and another one for the VGA modes. Determine the display characteristics of the XGA monitor for each instance.

4. Select one of the three possible memory apertures.
5. Determine memory size and availability of high-resolution modes. Select the bits-per-pixel resolution and the little-endian or big-endian data formats.
6. Select and enable an XGA mode.
7. Initialize the XGA coprocessor and determine and store its base address.
8. Determine and store the base address of the XGA video system.

## Locating the XGA Hardware

The first initialization task consists of locating the XGA components in the system's space. The necessary information is found in the PS/2 Programmable Option Select (POS) registers. Figure 8.6 shows important POS data related to the XGA hardware.

The first step in reading the POS registers is determining where these registers are located. In a micro channel IBM microcomputer BIOS service number 196, subservice number 0, of INT 15H returns the POS registers' base address in the DX register. The following code fragment shows the required processing for an XGA micro channel machine:

```
;****************************************************************
;       data variables for XGA initialization information
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_POS         DW      0       ; Base address of POS registers
; Contents of POS registers
```

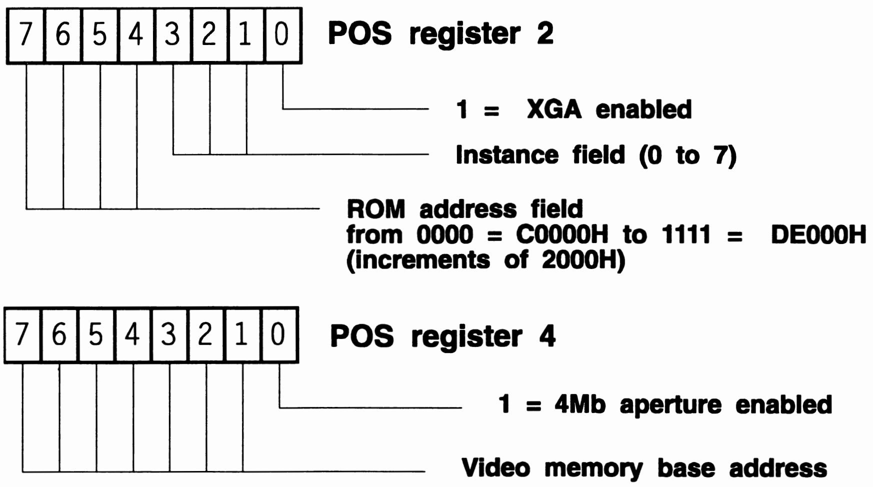


Figure 8.6 *Data in XGA POS Registers*

```
POS_2           DB      0       ; POS register at offset 2
POS_4           DB      0       ; POS register at offset 4
; Other data
XGA_REG_BASE    DW      0       ; Register base for XGA system
MON_ID          DW      0       ; Four-digit monitor ID code
                                ; obtained by new method
EQUIPMENT       DB      0       ; XGA equipment in system
; bitmap as follows:
;          7 6 5 4 3 2 1 0
;          | | | | | | | |___ 1 = XGA in system
;          | | | | | | |      0 = no XGA found
;          | | | | | | |_____ 1 = XGA color monitor
;          | | | | | |        0 = XGA monochrome monitor
;          | | | | | |_______ 1 = high-resolution (1024 x 768)
;          | | | | |          0 = no high-resolution
;          | | | | |_________ 1 = RAM = 1Mb
;          | | | |            0 = RAM = 512Kb
;          | | | |___________ 1 = dual monitor system
;          | | |              0 = single monitor system
;          |_|_|_____________ UNUSED
MODE            DW      0       ; Mode number during init
;
; Storage for 4 data bytes read from the Display ID register
VAR_A   DB      0               ; First byte read
VAR_B   DB      0
VAR_C   DB      0
VAR_D   DB      0               ; Last byte read
; Storage for 4 monitor ID nibbles
NIB_3   DB      0               ; First nibble to merge
NIB_2   DB      0
NIB_1   DB      0
NIB_0   DB      0
;
XGA_DIRECT      ENDS
;****************************************************************
;       processing operations for XGA initialization
;****************************************************************
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE
;
.386
;
INIT_XGA        PROC    FAR
; The .386 directive for Microsoft MASM allows coding using the
; 80386/486 instruction set. Since present implementations of XGA
; require the 30386 or higher processor, the use of these
; .386 instruction is valid
;**********************|
; set DS to XGA_DIRECT |
;**********************|
        MOV     AX,XGA_DIRECT           ; Local data segment
        MOV     DS,AX            ; to DS
        ASSUME  DS:XGA_DIRECT
; At this point the DS segment register can address data in the
; segment named XGA_DIRECT
;*********************|
;   get POS address   |
;*********************|
; Use service number 196, INT 15H, with AL = 0 to determine base
; address of Programmable Option Select (POS) registers
        MOV     AX,0C400H       ; AH = C4H (service request)
                                ; AL = 0 (subservice)
        INT     15H             ; BIOS interrupt
                                ; for micro channel machines only
        JNC     VALID_POS       ; Go if POS address returned
        JMP     NO_XGA          ; Error - not micro channel
VALID_POS:
        MOV     XGA_POS,DX   ; Save base address of POS
; An XGA system can be located on the motherboard or in one
; of 9 possible slots. Initialize CX = 0 for motherboard XGA
; CX = 1 to 9 for XGA in adapter card
        XOR     CX,CX           ; Start with motherboard
        CLI                     ; Interrupts off
; ... continues in the following code listing
```

Notice that not all POS values encode XGA data. The valid range for XGA systems is 8FD8H to 8FDBH. Service number 196, subservice number 1, of INT 15H can be used to enable each one of eight possible slots for setup. Then the value stored at the POS register base is read and compared to the valid range. If the value is within the range, an XGA adapter or motherboard implementation has been detected. In this case the POS registers contain data required for the initialization of the XGA system. The following code fragment illustrates the required processing:

```
; Use BIOS service 196, subservice number 1, to enable slot
; for setup
GET_POS_0:
        MOV     AH,0C4H         ; BIOS service
        MOV     AL,01H          ; Subservice number
        MOV     BX,CX           ; Slot number to BX
        INT     15H
; Slot enabled for setup
        MOV     DX,XGA_POS      ; POS register 0 and 1
        IN      AX,DX           ; Read ID low and high bytes
; Valid range for XGA systems is 8FD8H to 8FDBH
        CMP    AX,08FD8H        ; Test low limit
        JAE    TEST_HIGH_LIM    ; Go if equal or greater
; At this point the POS reports that system is not an XGA
; adapter
NOT_XGA_POS:
        INC     CX              ; CX is options counter
        CMP     CX,9            ; Done all slots?
        JB      GET_POS_0       ; Go if not at last slot
        JMP     NO_XGA          ; No XGA exit
TEST_HIGH_LIM:
        CMP     AX,08FDBH       ; Test high limit of range
        JA      NOT_XGA_POS     ; Go if out of range
;*********************|
;     XGA found       |
;*********************|
        CLI                     ; Disable interrupts
; Test if XGA is in motherboard
        CMP     CX,0            ; 0 is motherboard value
        JNE     XGA_CARD        ; Go if not on the motherboard
;*********************|
;  motherboard XGA    |
;*********************|
; Port 94H is used to enable and disable motherboard video
        MOV     AL,0DFH         ; Bit 5 = 0 for video setup
        MOV     DX,94H          ; 94H is system board enable
        OUT     DX,AL
        JMP     SHORT GET_POS   ; Skip slot setup
;*********************|
;       XGA card      |
;*********************|
XGA_CARD:
        MOV     AX,0C401H       ; Place adapter in setup mode
        MOV     BX,CX           ; Slot number to BL
        INT     15H
;*********************|
;  save POS registers |
;*********************|
GET_POS:
        MOV     DX,XGA_POS      ; Get POS record for the slot id
        ADD     DX,2            ; POS register at offset 2
        IN      AL,DX           ; Read data byte
        MOV     POS_2,AL        ; and store it
        INC     DX              ; Next POS register
        INC     DX              ; is number 4
        IN      AL,DX           ; Get contents
        MOV     POS_4,AL        ; Store it
; At this point POS registers 2 and 4 have been saved in
; variables
;*********************|
;   re-enable video   |
;*********************|
; Test for XGA in motherboard
        CMP     CX,0            ; Treat the motherboard
                                ; differently
        JNE     XGA_ADAPTER     ; Go if not in motherboard
; XGA in motherboard. Set bit 5 in port 94H to reenable video
        MOV     AL,0FFH         ; All bits set
        OUT     094H,AL
        JMP     SHORT REG_BASE
XGA_ADAPTER:
        MOV     AX,0C402H          ; Enable the slot for normal
        MOV     BX,CX              ; operation
        INT     15H
; ... continues in the following code listing
```

## Instance and Register Base Address

The next step in the XGA initialization is calculating the XGA Display Controller register base by adding the instance value to the address template 21x0H previously mentioned. The following code shows the necessary manipulation of the instance bits:

```
; ... continues from the previous code listing
;**********************|
; calculate and store  |
; XGA register base    |
;**********************|
REG_BASE:
        STI                       ; Interrupts on again
        MOV     AL,POS_2          ; Get value at POS register 2
        AND     AX,0EH            ; Mask out all bits except
                                  ; instance
        SHL     AX,1              ; Move instance value to second
        SHL     AX,1              ; digit position
        SHL     AX,1
        ADD     AX,2100H          ; Add instance to base address
        MOV     XGA_REG_BASE,AX   ; Store result in variable
; ... continues in the following code listing
```

## Obtaining Monitor ID Code

The IBM documentation describes two ways for determining the monitor hardware installed in an XGA system. The first and older method is by reading the display identification field (bits 0 to 3) in the Display ID and Comparator register. A new method for reading the display ID is described in the *Personal System /2 Hardware Interface Technical Reference – Video Susbsystems*, published by IBM in September 1992. (See Bibliography.) The data obtained by means of the new method is in the form of four hexadecimal digits. This last format coincides with the monitor ID reported by the DMQS function, discussed later in this chapter.

In the initialization code that follows we used both methods of monitor identification. The older method is used to determine if the system is equipped with a color or monochrome monitor and if high-resolution hardware (1024-by-768 pixels) is available. Since this information may not be valid for new IBM monitors, the code also reports the monitor ID bits in the format recommended by IBM. Software should base its mode setting decisions on the new 4-digit monitor ID codes, rather than on the older 4-bit values. Table 8.5 shows the bit codes and characteristics of several popular monitors in both formats. Note that for some monitors listed in Table 8.5 no 4-bit code is listed since this information has not been documented.

Table 8.5 *XGA Display ID Codes*

| MONITOR ID | | DISPLAY | | | MAXIMUM DIMENSIONS/COLORS OR GRAY | |
|---|---|---|---|---|---|---|
| 4-bit | 4-digit | MODEL | (IN.) | TYPE | 512K | 1MB |
| 1111 | FFFF | none | -- | ---------- | ------------- | ------------- |
| 1101 | FF0F | 8503 | 12 | monochrome | 640-by-480/64 | 640-by-480/64 |
| 1110 | FFF0 | 8513 | 12 | color | 640-by-480/256 | 640-by-480/256 |
| | | 8512 | 14 | | 640-by-480/65536 | |
| ---- | | 8518 | | | | |
| 1011 | F0FF | 8515 | 14 | color | 640-by-480/256 | 640-by-480/65536 |
| ---- | | 8516 | | | 1024-by-768/16 | 1024-by-768/256 |
| 1001 | F00F | 8604 | 15 | monochrome | 640-by-480/64 | 1024-by-768/64 |
| | | 8507 | 19 | monochrome | 1024-by-768/16 | 1024-by-768/64 |
| 1010 | F0F0 | 8514 | 16 | color | 640-by-480/256 | 640-by-480/65536 |
| | | | | | | 1024-by-768/256 |
| 0010 | F0F0 | 8514 | 16 | color | 640-by-480/256 | 640-by-480/65536 |
| | | | | | | 1024-by-768/256 |
| ---- | 90F0 | 8517 | 17 | color | 1024-by-768/256 | 1024-by-768/256 |

Obtaining the 4-bit original monitor identification code is straightforward and uncomplicated. Software simply selects the Display ID and Comparator register, which is the indexed-access register at offset 52H, then reads its contents by means of an IN instruction. The monitor ID field is the low-order nibble thus obtained. The following code fragment shows the processing operations:

```
; ... continues from the previous code listing
;**********************|
;   get monitor code   |
;   in 4-bit format    |
;**********************|
; Bits 0 to 3 of Display ID and Comparator register (offset 52H)
; encode the display type attached to the system
        MOV     DX,XGA_REG_BASE ; Base address
        ADD     DX,0AH          ; To Index register
        MOV     AL,052H         ; Select Display ID and
                                ; Comparator register
        OUT     DX,AL
; Read byte at selected register
        MOV     DX,XGA_REG_BASE
        ADD     DX,0BH           ; To data port
        IN      AL,DX            ; Read register data
        AND     AL,0FH           ; Mask off high nibble
        CMP     AL,00H           ; Test for illegal value
        JNE     ID_OK           ; Go if valid display ID
; Illegal ID code
        JMP     NO_XGA
;**********************|
;    monitor ID bits   |
;**********************|
ID_OK:
; Monitor ID is reported as follows:
; bit code    color/BW  max. address    Model
; xxxx 1111 = no display installed
; xxxx 1101 = color     640-by-480      8503
; xxxx 1110 = color     640-by-480      8512/8513
; xxxx 1011 = color     1024-by-768     8515
; xxxx 1001 = BW        1024-by-768     8504/8507
; xxxx 1010 = color     1024-by-768     8514
; xxxx 0010 = color     1024 by 768     8514
;        ||
;        ||------- Bit 1 = 1 for color displays
;        |
;        |-------- Bit 2 = 0 in displays that
;                          support high-resolution modes
;
;*********************|
;    set monitor bit   |
;*********************|
        AND     AL,00000110B     ; Preserve bits 1 and 2
                                 ; and clear bit 0
        XOR     AL,00000101B     ; Reverse bits 0 and 2
        MOV     EQUIPMENT,AL     ; Save the result flag
; Read bit 0 of the Operating Mode register (offset 0) to
; determine if VGA mode address decoding is enabled
        MOV     DX,XGA_REG_BASE      ; Operating Mode register
        IN      AL,DX           ; Read data byte
        TEST    AL,1            ; Test low bit
        JNZ     INT_CONTROL     ; Go single monitor in system
        OR      EQUIPMENT,00010000B     ; Set bit 5 to indicate
                                        ; 2 monitors
; ... continues in the following code listing
```

Obtaining the 4-digit monitor ID code that is consistent with the value reported by the DMQS service is a considerably more complicated task. For this reason, if the software determines that the system supports DMQS, this is the preferred way for reading the monitor ID and other system data. However, since the XGA adapters built before the XGA-2 upgrade do not support DMQS, it is convenient to have available a method for reading the 4-digit monitor ID codes.

The information necessary for the 4-digit monitor ID code is stored in the Display ID and Comparator register, at offset 52H of the XGA indexed-access register group. In this case the data necessary for determining the four hexadecimal monitor ID digits is obtained by performing four successive read operations to this register. The process requires the following manipulations:

1. The XGA CRT Controller is prepared for reset by writing 01 binary to the Display Blanking field (bits 0 and 1) of the Display Controller 1 register at offset 50H of the indexed-access register group.
2. The CRTC Controller is reset by writing 00 binary to the same Display Blanking field of the Display Controller 1 register.

3. The Sync Polarity field (bits 6 and 7) of the Display Controller 1 register is set to 01 binary. After a 15-microsecond delay, data nibble A is read from the 4 low-order bits of the Display Comparator register at offset 52H of the indexed-access register group.
4. The Sync Polarity field of the Display Controller 1 register is set to 10 binary. After a 15-microsecond delay, data nibble B is read from the 4 low-order bits of the Display Comparator register.
5. The Sync Polarity field of the Display Controller 1 register is set to 00 binary. After a 15-microsecond delay, data nibble C is read from the 4 low-order bits of the Display Comparator register.
6. The Sync Polarity field of the Display Controller 1 register is set to 11 binary. After a 15-microsecond delay, data nibble D is read from the 4 low-order bits of the Display Comparator register.

The four data nibbles read contain the 4-digit monitor ID code, as shown in Figure 8.7. Notice in Figure 8.7 that the four monitor ID digits are derived from the corresponding bits in the four data nibbles. In other words, all bits number 3 are collected to form the third digit, all bits number 2 are collected to form the second digit, and so forth. The assembled monitor ID code is a word-size unit containing 4 nibbles, each nibble encoding one ID digit. The processing for obtaining the four data nibbles and for assembling the monitor ID digits is shown in the following code fragment:

```
; ... continues from the previous code listing
;**********************|
;  obtain monitor ID   |
;       digits         |
;**********************|
; Prepare CRTC for reset by writing binary 01 to the XGA Display
```

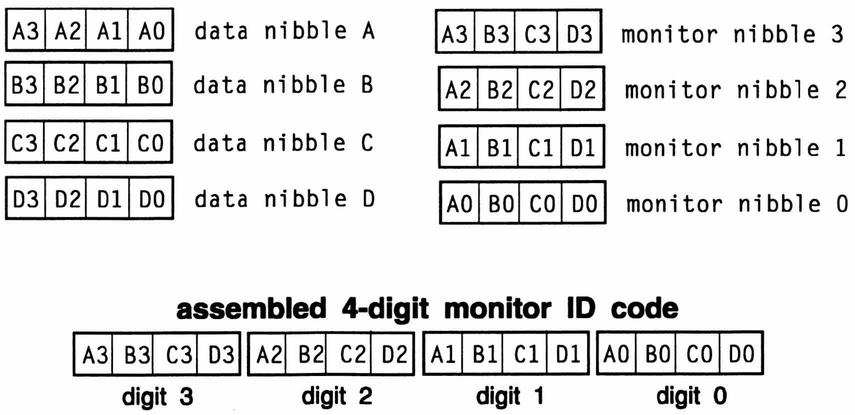


Figure 8.7 *XGA Monitor ID Code*

```
        POP     DS              ; Restore caller's DS
        RET
SPRITE_AT       ENDP
P_CODE          ENDS
```

## Turning Off the Sprite

At times the software needs to erase the sprite image from the video display. This is achieved by clearing bit 0 of the Sprite Control register at offset 36H. The following procedure turns off the sprite:

```
;****************************************************************
;        processing operations for erasing the sprite
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
SPRITE_OFF      PROC    FAR
; Sprite is turned off by clearing bit 0 of the Sprite Control
; register at offset 36H
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT   ; Local data segment
        MOV     DS,AX           ; to DS
        ASSUME  DS:XGA_DIRECT
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Sprite Control register offset is 36H
        MOV     AH,0            ; Value to start register
        MOV     AL,36H          ; Address of Start register
        OUT     DX,AX           ; Write data
        POP     DS              ; Restore caller's DS
        RET
SPRITE_OFF      ENDP
P_CODE          ENDS
```

```
        ADD     DX,0AH          ; To Index register
; Index register 30H is Sprite x Start LOW register
        MOV     AH,BL           ; Value to start register
        MOV     AL,30H          ; Address of Start x Low
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 31H is Sprite x Start HIGH register
        MOV     AH,BH           ; Value to start register
        MOV     AL,31H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Set Sprite x Preset register to 0
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 32H is Sprite x Preset register
        MOV     AH,00           ; Value to preset register
        MOV     AL,32H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Select y coordinate registers
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To index register
; Index register 33H is Sprite y Start LOW register
        MOV     AH,CL           ; Value to start register
        MOV     AL,33H          ; Address of Start x Low
        OUT     DX,AX           ; Write data
;
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 34H is Sprite y Start HIGH register
        MOV     AH,CH           ; Value to start register
        MOV     AL,34H          ; Address of Start register
        OUT     DX,AX           ; Write data
; Set Sprite y Preset register to 0
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 35H is Sprite x Preset register
        MOV     AH,00           ; Value to preset register
        MOV     AL,35H          ; Address of Start register
        OUT     DX,AX           ; Write data
;**********************|
;   display sprite     |
;**********************|
; Sprite is displayed by setting bit 0 of the Sprite Control
; register at offset 36H
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
; Sprite control register offset is 36H
        MOV     AH,01           ; Value to start register
        MOV     AL,36H          ; Address of Start register
        OUT     DX,AX           ; Write data
        OUT     DX,AX            ; Send to data port
        INC     SI               ; Bump data pointer
        INC     SI               ; to next word
        LOOP    SPRITE_DATA      ; Repeat 512 times
        POP     BP               ; Restore caller's BP
        RET
SPRITE_IMAGE    ENDP
P_CODE          ENDS
```

### 9.8.4 Sprite Manipulations

Sprite display operations consist of turning on the sprite image at a predetermined screen address and of erasing the sprite. Note that when the sprite is turned on at a new position, the old sprite image is automatically erased by the hardware without disturbing the underlying screen pixels. Therefore moving the sprite on the XGA screen does not require erasing the existing sprite image or saving the underlying pixels.

#### Turning On the Sprite

As previously mentioned, if the low-order bit of the Sprite Control register is set, the sprite image is displayed on the video screen. The position at which it is displayed is determined by the setting of the Sprite Horizontal Start and Vertical Start registers. The following procedure can be used to display a previously loaded sprite image at a screen location supplied by the caller:

```
;****************************************************************
;       processing operations for displaying the sprite
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
;
SPRITE_AT       PROC    FAR
; Display a previously loaded sprite image at coordinates
; furnished by the caller
; On entry:
;           BX = x coordinate of sprite location
;           CX = y coordinate of sprite location
; Screen range is according to active mode
; Sprite preset values are 0
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT    ; Local data segment
        MOV     DS,AX            ; to DS
        ASSUME  DS:XGA_DIRECT
        MOV     DX,XGA_REG_BASE ; Register base
        OUT     DX,AX           ; Write data
;
        MOV     DX,BP           ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 3CH is Sprite Color 1, green value
        MOV     AL,3CH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;
        MOV     DX,BP           ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 3DH is Sprite Color 1, blue value
        MOV     AL,3DH          ; Sprite register
        MOV     AH,[SI]         ; Data from caller's buffer
        INC     SI              ; Bump pointer to next byte
        OUT     DX,AX           ; Write data
;**********************|
;  load sprite image   |
;**********************|
; First set the Sprite Index registers to 0
        MOV     DX,BP           ; Register base
        ADD     DX,0AH          ; To index register
; Index register 60H is Sprite/Palette index Low
        MOV     AX,0060H        ; 00 to register at offset 60H
        OUT     DX,AX           ; Write data
; Reset to base
        MOV     DX,BP           ; Register base
        ADD     DX,0AH          ; To Index register
; Index register 61H is Sprite/Palette index High
        MOV     AX,0061H        ; 00 to register at offset 60H
        OUT     DX,AX           ; Write data
; Select the Sprite Data register at offset 6AH
        MOV     DX,BP           ; Register base
        ADD     DX,0AH          ; To Index register
        MOV     AL,06AH         ; Offset of data register
        OUT     DX,AL           ; Select the Sprite Data register
;**********************|
;  load sprite image   |
;**********************|
; DS:SI => buffer area containing the sprite bit-mapped image in
;           2-bits-per-pixel format, as follows:
;           00 = sprite color 0
;           01 = sprite color 1
;           10 = transparent pixel
;           11 = complement pixel
        MOV     CX,512          ; Word item counter
SPRITE_DATA:
        MOV     DX,BP           ; Register base
        ADD     DX,0CH          ; To second data register
        MOV     AX,[SI]         ; Get data from buffer
```

```
;
;*********************|
;   save caller's DS  |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    BP                ; Save caller's base pointer
        PUSH    DS
        MOV     AX,XGA_DIRECT     ; Local data segment
        MOV     DS,AX             ; to DS
        ASSUME  DS:XGA_DIRECT
        MOV     DX,XGA_REG_BASE ; XGA register base address
        MOV     BP,DX             ; Store in BP
        POP     DS                ; Restore caller's DS
;*********************|
; set sprite color 0   |
;*********************|
; Load Sprite Color 0 registers using values in parameter block
; supplied by caller (DS:SI)
        MOV     DX,BP             ; Register base
        ADD     DX,0AH            ; To Index register
; Index register 38H is Sprite Color 0, red value
        MOV     AL,38H            ; Sprite register
        MOV     AH,[SI]           ; Data from caller's buffer
        INC     SI                ; Bump pointer to next byte
        OUT     DX,AX             ; Write data
;
        MOV     DX,BP             ; Register base
        ADD     DX,0AH            ; To Index register
; Index register 39H is Sprite Color 0, green value
        MOV     AL,39H            ; Sprite register
        MOV     AH,[SI]           ; Data from caller's buffer
        INC     SI                ; Bump pointer to next byte
        OUT     DX,AX             ; Write data
;
        MOV     DX,BP             ; Register base
        ADD     DX,0AH            ; To Index register
; Index register 3AH is Sprite Color 0, blue value
        MOV     AL,3AH            ; Sprite register
        MOV     AH,[SI]           ; Data from caller's buffer
        INC     SI                ; Bump pointer to next byte
        OUT     DX,AX             ; Write data
;*********************|
; set sprite color 1  |
;*********************|
; Load Sprite Color 1 registers to GREEN
        MOV     DX,BP             ; Register base
        ADD     DX,0AH            ; To Index register
; Index register 3BH is Sprite Color 1, red value
        MOV     AL,3BH            ; Sprite register
        MOV     AH,[SI]           ; Data from caller's buffer
        INC     SI                ; Bump pointer to next byte
```

```
DATA     SEGMENT
;**********************|
;    sprite data       |
;**********************|
; The 64-by-64 pixel sprite is defined at 64 lines of 4
; doublewords per line
;
; First 6 bits of the sprite color are significant
; In this example color number 0 is bright red and color number 1
; is bright white
SPRITE_MAP_0    DB      11111100B       ; Red for color 0
                DB      0               ; Green for color 0
                DB      0               ; Blue for color 0
                DB      11111100B       ; Red for color 1
                DB      11111100B       ; Green for color 1
                DB      11111100B       ; Blue for color 1
; The 64-by-64 pixel sprite is defined as 64 lines of 4
; doublewords per line, encoded as follows:
; 00H = 00 00 00 00 B = 4 pixels in sprite color 0
; 55H = 01 01 01 01 B = 4 pixels in sprite color 1
; AAH = 10 10 10 10 B = 4 transparent pixels
; FFH = 11 11 11 11 B = 4 pixels in one's complement of image
;
                DD      256 DUP (0055AAFFH)
DATA      ENDS
```

The following procedure can be called by an application to load a sprite image formatted as shown in the previous code fragment.

```
;*****************************************************************
;      processing operations for loading a sprite image
;*****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
;
SPRITE_IMAGE    PROC    FAR
; Load sprite image and select color registers
; On entry:
;         DS:SI => caller's sprite image buffer and color buffer
;                  formatted as follows:
;   OFFSET    UNIT      CONTENTS
;     0       byte      6 low bits are RED for sprite color 0
;     1       byte      6 low bits are GREEN for sprite color 0
;     2       byte      6 low bits are BLUE for sprite color 0
;     3       byte      6 low bits are RED for sprite color 1
;     4       byte      6 low bits are GREEN for sprite color 1
;     5       byte      6 low bits are BLUE for sprite color 1
;     6       16 bytes per 64 rows (1024 bytes) encoding the
;             sprite image at 2 bits per pixel
;   1030      end of sprite image
```

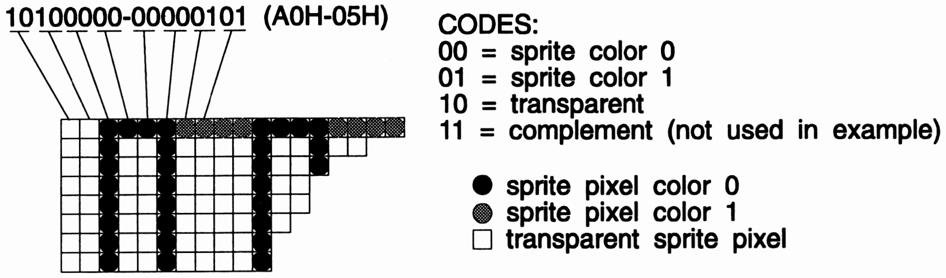


Figure 9.13 *Sample of Sprite Image Bitmap*

## 9.8.2 Sprite Colors and Attributes

The sprite's attributes are encoded into a 2-bit field. The first two codes refer to sprite color attributes, the third code defines a transparent attribute, and the last one defines a one's complement operation. (See Table 9.7.) The sprite colors 0 and 1 are determined by the settings in two sets of registers in the Display Controller group: registers 38H to 3AH select the red, green, and blue values of sprite color 0, while registers 3BH to 3DH select the same values in sprite color 1. In this manner, if the first byte in the sprite buffer is encoded with the value 01010101B, then the first four bits in the sprite are displayed using the color value for sprite color 1. Figure 9.13 shows how the sprite pixels are mapped to the binary values stored in the sprite buffer.

In summary, the attribute of each sprite pixel corresponds to the two-bit code stored in the sprite buffer. Therefore, designing a sprite image is a matter of installing the red, green, and blue values for each sprite color and then composing a pixel map using the two-bit values in Table 9.7. The Sprite Horizontal and Vertical Preset registers can be used to adjust a sprite image that does not coincide with the top-left corner of the map stored in the sprite buffer.

## 9.8.3 Loading the Sprite

Once the sprite map has been composed and stored in an application's memory variable, the software can proceed to set the Sprite Color registers and load the image into the sprite buffer. The following code fragment assumes that the sprite colors and bitmap have been placed in a formatted parameter block. From this data the sprite color values and image are loaded into the corresponding Display Controller registers.

For example, the following code fragment shows a sprite image defined in the application's data area. The sprite data includes the values for both sprite colors.

## 9.8.1 The Sprite Image

The sprite image consists of a 64-by-64 pixel bitmap. Each sprite image pixel can have one of four attributes. The storage structure is in Intel data format and encoded in 2 bits per pixel. The bit codes for the sprite image are shown in Table 9.7.

Table 9.7 *Sprite Image Bit Codes*

| BIT CODE | ACTION |
|---|---|
| 00 | Pixel displayed in sprite color 0 |
| 01 | Pixel displayed in sprite color 1 |
| 10 | Transparent (image pixel is visible) |
| 11 | Complement (one's complement of image pixel is visible) |

The displayed sprite can be smaller than 64-by-64 pixels. In this case the software controls, by means of the Sprite Horizontal Preset (offset 32H) and Sprite Vertical Preset registers (offset 35H) in the Display Controller, which part of the sprite image is displayed. However, the sprite image always extends to the full 64-bit length and width of the sprite buffer. Transparent sprite codes can be used to locate the sprite image within the pixel rectangle defined by the 64-byte sprite buffer. The elements used in controlling the size of the sprite image are shown in Figure 9.12.

The location of the sprite image within the viewport is determined by the Sprite Horizontal Start and Sprite Vertical Start registers. Both of these registers are word-size; however, the valid range of values is limited to 0 to 2047. The low-order bit in the Sprite Control register (offset 36H) determines the sprite's visibility. The sprite is displayed when this bit is set and is invisible if this bit is cleared.

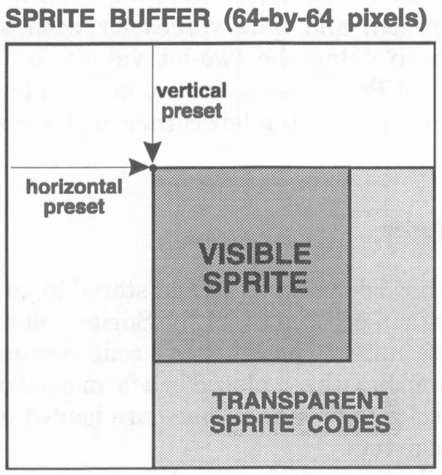


Figure 9.12 *Sprite Image Controls*

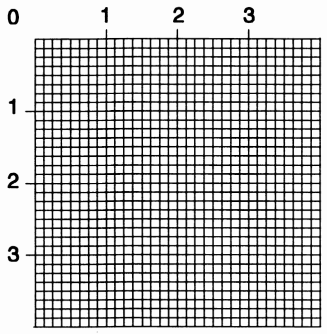


Figure 9.11 *XGA Sprite Buffer*

The XGA sprite mechanism consists of hardware elements designed to store and display a small graphics object. The sprite operation is independent of the video display function. The maximum size of the sprite image is 64-by-64 pixels. This image is stored in a 32K static RAM chip (which is not part of video memory) called the *sprite buffer*. This buffer is used for storing alphanumeric characters when XGA is in a VGA mode or in its proprietary 132-column text mode. The main advantage of the XGA sprite is that it does not affect the image currently displayed; therefore, the XGA programmer need not worry about preserving the video image as the sprite is moved on the screen. This action can be best visualized as a transparent overlay that is moved over the picture without changing it. Figure 9.11 shows the XGA sprite buffer.

The XGA registers related to sprite image display and control are located in the Indexed Access register group of the Display Controller. Table 9.6 lists the location and purpose of the sprite-related registers.

Table 9.6 *Sprite Registers in the Display Controller*

| INDEX REGISTER OFFSET | REGISTER NAME |
|---|---|
| 30H | Sprite horizontal start, low part |
| 31H | Sprite horizontal start, high part |
| 32H | Sprite horizontal preset |
| 33H | Sprite vertical start, low part |
| 34H | Sprite vertical start, high part |
| 35H | Sprite vertical preset |
| 36H | Sprite control register |
| 38H | Sprite color 0, red component |
| 39H | Sprite color 0, green component |
| 3AH | Sprite color 0, blue component |
| 3BH | Sprite color 1, red component |
| 3CH | Sprite color 1, green component |
| 3DH | Sprite color 1, blue component |
| 60H | Sprite/palette index, low part |
| 61H | Sprite/palette index, high part |
| 62H | Sprite/palette prefetch, low part |
| 63H | Sprite/palette prefetch, high part |
| 6AH | Sprite data |
| 6BH | Sprite prefetch save (RESERVED) |

```
        MOV     GS:[+60H],CX     ; Write to Operation Dimension 1
                                 ; register
; Then calculate Term E
        PUSH    DX               ; Save |y|
        ADD     DX,DX            ; 2 * |y|
        SUB     DX,CX            ; - |x|
        MOV     SI,DX            ; Store Term E in SI
        POP     DX               ; Restore |y|
        PUSH    CX               ; and save |x|
        MOV     CX,DX            ; |y| to CX
        ADD     CX,CX            ; Calculate 2 * |y|
        MOV     DI,CX            ; Store Term K1 in DI
        POP     CX               ; Restore |x| from stack
        SUB     DX,CX            ; |y| - |x|
        ADD     DX,DX            ; times 2
; DX = Term K2
        MOV     GS:[+20H],SI     ; Write to Error Term register
        MOV     GS:[+24H],DI     ; Write to K1 register
        MOV     GS:[+28H],DX     ; Write to K2 register
; Bitmap of Pixel Operations register:
; byte 3 = 0000|0101 = line draw write operation
; byte 2 = 0001      = source pixel map is map A
;               0001 = destination pixel map is map A
; byte 1 = 1000|rrrr = special code for foreground and all 1's
; byte 0 = 00   0    = Map mask disabled
;            00      = Drawing mode for all pixels drawn
;                      OCTANT DATA:
;                o   = DX = 0 for x in positive direction
;                 o  = DY = 0 for y in positive direction
;                  o = DZ = 0 for |x| > |y|
        MOV     EAX,05118000H    ; All bits except octant
; BL holds octant bits
        OR      AL,BL            ; OR-in octant bits
        MOV     GS:[+7CH],EAX    ; Write to Pixel Operations
                                 ; register
        RET
COP_LINE_2      ENDP
P_CODE          ENDS
```

## 9.8 Programming The XGA Sprite

Many graphics programs, at both the system and the application levels, must manipulate some sort of animated screen marker image. A typical example of a screen marker is a mouse-controlled pointer or icon often used to select from option boxes or menus. Since the marker image overlays the screen, in previous graphics systems software had to find ways for saving and restoring the screen contents as this image was translated over the pixel grid. However, in XGA the operation of a small screen pointer icon is considerably simplified thanks to a hardware-supported device called the sprite.

```
;*********************|
;  calculate octant 0  |
;*********************|
;        CX = x pixel coordinate of line start
;        DX = y pixel coordinate of line start
;        SI = x pixel coordinate of line end
;        DI = y pixel coordinate of line end
; Octant bits in Pixel Operations register as follows:
; xxxx x210
;       |||_______ DZ bit = 0 if |x| > |y|
;       ||________ DY bit = 0 if y is positive (DI = DX)
;       |_________ DX bit = 0 if x is positive (SI = CX)
;
; BL will hold octant bits
        MOV     BL,0            ; Clear octant register
        CMP     SI,CX           ; Test for DX bit
        JGE     DX_ISOK         ; Go if horizontal line
; At this point SI  CX, therefore DX bit must be set
        OR      BL,00000100B    ; DX bit is now set in BL
        XCHG    SI,CX           ; Exchange so that CX  SI
DX_ISOK:
; Now test DX bit condition
        CMP     DI,DX           ; Test for DY bit
        JGE     DY_ISOK         ; Go if horizontal line
; At this point DI  DX, therefore DY bit must be set
        OR      BL,00000010B    ; DY bit is now set in BL
        XCHG    DI,DX           ; Exchange so that DX  DI
; Now test DX bit condition
DY_ISOK:
        SUB     DI,DX           ; Find |y|
        XCHG    DX,DI           ; |y| to DX
        SUB     SI,CX           ; and |x|
        XCHG    CX,SI           ; |x| to CX
        CMP     CX,DX           ; Is |x| > |y|
        JG      BRZ_TERMS       ; Go to leave DZ = 0
; At this point |x| > |y|, therefore DZ bit must be set
; and |y| must be exchanged with |x|
        OR      BL,00000001B    ; Set DZ bit
        XCHG    CX,DX           ; Exchange
;
;*********************|
;   Bresenham terms    |
;     calculations     |
;*********************|
BRZ_TERMS:
; Bresenham terms:
;     Term E (error) = (2 * |y|) - |x|
;     Term K1        = 2 * |y|
;     Term K2        = 2 * (|y| - |x|)
; AT this point CX = |x| and DX = |y|
; First store |x| in Operations Dimensions register
; Draw line using XGA graphics coprocessor
; Code assumes:
;       1. 1024-by-768 mode in 256 colors
;       2. The INIT_COP routine has been previously called to
;          initialize the GS and FS segment registers
; On entry:
;        CX = x pixel coordinate of line start
;        DX = y pixel coordinate of line start
;        SI = x pixel coordinate of line end
;        DI = y pixel coordinate of line end
;        BL = 8-bit color code
;*********************|
;   test for not busy |
;*********************|
        CALL    COP_RDY         ; Local routine
; At this point the coprocessor is not busy
;
;*********************|
;   prepare to draw   |
;*********************|
; Draw line accessing coprocessor registers directly
        MOV     AL,01H          ; Data value for Map A
        MOV     GS:[+12H],AL    ; Write to pixel map index
        MOV     AX,0H           ; Data value for VRAM low
        MOV     GS:[+14H],AX    ; Write to pix map base address
; FS register holds the high-order word of VRAM address. This
; value is calculated by the INIT_COP routine in this module
        MOV     AX,FS           ; Data for VRAM high
        MOV     GS:[+16H],AX    ; Write to pix map segment
                                ; address
; Code assumes 1024-by-768 pixel mode and Intel format
        MOV     AX,1023         ; Value for pix map width
        MOV     GS:[+18H],AX    ; Write to width register
        MOV     AX,767          ; Value for pix map height
        MOV     GS:[+1AH],AX    ; Write to height register
        MOV     AL,3            ; Select Intel order and
                                ; 8 bits per pixel
        MOV     GS:[+1CH],AL    ; Write to format register
;*********************|
;     draw the line   |
;*********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
; Write color (in BL) to Foreground register
        MOV     GS:[+58H],BL    ; Write to Foreground Color
                                ; register
; Write coordinates of line start point to coprocessor registers
        MOV     GS:[+78H],CX    ; Write to Destination x Address
                                ; register
        MOV     GS:[+7AH],DX    ; Write to Destination y Address
                                ; register
```

The following rules allow normalizing to the first octant any line defined by its start and end points:

1. If the end $x$ coordinate is smaller than the start $x$ coordinate, set the DX bit in the Pixel Operations register.
2. If the end $y$ coordinate is smaller than the start $y$ coordinate, set the DY bit in the Pixel Operations register.
3. If the difference between the $y$ coordinates is greater than or equal to the difference between the $x$ coordinates, set the DZ bit in the Pixel Operations register.
4. After the octant bits DX, DY, and DZ are set according to the above rules, the code can use the unsigned difference between $y$ coordinates (delta $y$ or D$y$) and the unsigned difference between $x$ coordinates (delta $x$ or D$x$) in the remaining calculations.

## 9.7.2 Calculating the Bresenham Terms

Three coprocessor registers are used to encode values that result from applying Bresenham's algorithm; these are the Bresenham Error Term register (offset 20H), the Bresenham K1 Term register (offset 24H), and the Bresenham K2 Term register (offset 28H).

The Bresenham K1 constant is calculated by the formula:

$$\text{Term K1} = 2 * \text{D}y$$

Recall that D$y$ is the absolute difference between $y$ coordinates, and D$x$ the absolute difference between $x$ coordinates. The Bresenham K2 constant is calculated by the formula:

$$\text{Term K2} = 2 * (\text{D}y - Dx)$$

The Bresenham error term is calculated by the formula:

$$\text{Term E} = (2 * \text{D}y) - \text{D}x$$

The Bresenham terms are entered into the corresponding coprocessor registers. (See Table 9.2.) The Operation Dimension 1 register (at offset 60H) is loaded with the value of D$x$. The following code procedure shows the necessary processing for drawing a straight line using the XGA coprocessor:

```
;****************************************************************
;          processing operations for a Bresenham line draw
;****************************************************************
P_CODE    SEGMENT PUBLIC
          ASSUME  CS:P_CODE
.386
COP_LINE_2        PROC     FAR
```

## 9.7 Line Drawing Operations

The XGA draws a straight line following a method originally described by J. E. Bresenham (*IBM Systems Journal*, 1965) and since known as Bresenham's algorithm. Bresenham's method is based on the differential equation for the slope of a straight line, which states that the difference between the *y* coordinates divided by the difference between the *x* coordinates is a constant. This constant, usually called the slope, is conventionally designated by the letter *m*, as in the formula:

$$m = \frac{Dy}{Dx}$$

where D*y* is the difference between the *y* values and D*x* the difference between the *x* values. Therefore, *y* can be expressed as a function of *x*, as follows:

$$y = mx$$

Bresenham's algorithm, as implemented on XGA, requires that all parameters be normalized to the first octant (octant number 0). Figure 9.10 shows the octant numbering in the cartesian plane.

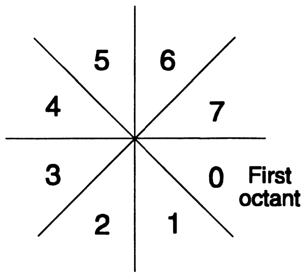


Figure 9.10 *XGA Numbering of Cartesian Octants*

### 9.7.1 Reduction to the First Octant

The octant is selected by the octant field bits in the Pixel Operations register. The 1-bit values designated DX, DY, and DZ have the following meaning:

1. DX encodes the direction of the *x* values in reference to the line's start point. DX = 0 if *x* is in the positive direction, and DX = 1 if it is in the negative direction.
2. DY encodes the direction of the *y* values in reference to the line's start point. DY = 0 if *y* is in the positive direction and DY = 1 if it is in the negative direction.
3. DZ encodes the relation between the absolute value of the *x* and *y* coordinates. DZ = 0 if $|x| > |y|$, and DZ = 1 otherwise.

```
;                      ss = foreground source
;                          *00 = foreground color
;                           10 = source pixel map
;                      pppp = function
;                           0010 = draw and step read
;                           0011 = line draw read
;                           0100 = draw and step write
;                           0101 = line draw write
;                          *1000 = pixBlt
;                           1001 = inverting pixBlt
;                           1010 = area fill pixBlt
;                       BYTE 3 = 00001000B = 08H
; byte 2 = SSSS|DDDD (* = values for this operation)
;                      SSSS = source pixel map
;                           0001 = pixel map A
;                          *0010 = pixel map B
;                           0011 = pixel map C
;                      DDDD = destination pixel map
;                          *0001 = pixel map A
;                           0010 = pixel map B
;                           0011 = pixel map C
;                       BYTE 2 = 00100001B = 21H
; byte 1 = PPPP|0000 (* = values for this operation)
;                      PPPP = pattern pixel map
;                           0001 = pixel map A
;                          *0010 = pixel map B
;                           0011 = pixel map C
;                           1000 = foreground (fixed)
;                           1001 = generated from source
;                       BYTE 1 = 00100000B = 20H
; byte 0 = mm00|0oox (* = values for this operation)
;                      mm = mask pixel map
;                          *00 = mask map disabled
;                           01 = boundary enabled
;                           10 = mask map enabled
;                      oox = octant bits (x = don't care)
;                          *00 = start at top left and move
;                                right and down
;                           10 = start at top right and move
;                                left and down
;                           01 = start at bottom left and move
;                                right and up
;                           11 = start at bottom right and move
;                                left and up
;                       BYTE 0 = 00000000B = 00H
        MOV     EAX,008212000H  ; Value from bitmap
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
        RET
COP_PATBLT      ENDP
P_CODE          ENDS
```

```
; Dimensions of source map are in CX and DX registers
        DEC     CX
        DEC     DX
        MOV     GS:[+18H],CX    ; Write to width register
        MOV     GS:[+1AH],DX    ; Write to height register
; Bitmap of pixel format register:
; 7 6 5 4 3 2 1 0 <= bits
; | | | | | |_|_|______ pixel image size (* = selected value)
; | | | | |            *000 = 1 bit per pixel
; | | | | |             001 = 2 bits per pixel
; | | | | |             010 = 4 bits per pixel
; | | | | |             011 = 8 bits per pixel
; | | | | |_______ format control
; | | | |          *1 = Motorola order
; | | | |           0 = Intel order
; |_|_|_|_______________________________ RESERVED
        MOV     AL,08H          ; Select Motorola order and
                                ; 1 bit per pixel
        MOV     GS:[+1CH],AL    ; Write to format register
;*********************|
;    select mix mode  |
;*********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
;*********************|
;     store color     |
;*********************|
; Write color (in BL) to Foreground register
        MOV     GS:[+58H],BL    ; Write to Foreground Color
                                ; register
; Write coordinates of source and destination
; Source coordinate are 0,0, destination coordinates are in SI
; and DI
        MOV     AX,0            ; Source coordinates
        MOV     GS:[+74H],AX    ; Write to Source x Address
        MOV     GS:[+76H],AX    ; Write to Source y Address
        MOV     GS:[+78H],SI    ; Write to Destination x Address
        MOV     GS:[+7AH],DI    ; Write to Destination y Address
; Store width in Operations Dimension 1 register
        MOV     GS:[+60H],CX    ; Write to Operation Dimension 1
; Store height in Operations Dimension 2 register
        MOV     GS:[+62H],DX    ; Write to Operation Dimension 2
;*********************|
;  setup pix operation |
;      registers      |
;*********************|
; Bitmap of Pixel Operations register for pixBlt operation:
; byte 3 = bbss|pppp (* = values for this operation)
;                    bb = background source
;                        *00 = background color
;                         10 = source pixel map
```

```
        CALL    COP_RDY          ; Local routine
; At this point the coprocessor is not busy
;**********************|
; map A is destination |
;   (video memory)     |
;**********************|
        PUSH    AX               ; Bitmap offset to stack
        MOV     AL,01H           ; Data value for Map A
        MOV     GS:[+12H],AL     ; Write to pixel map index
        MOV     AX,0H            ; Data value for VRAM low
        MOV     GS:[+14H],AX     ; Write to pix map base address
; FS register holds the high-order word of VRAM address. This
; value is calculated by the INIT_COP routine in this module
        MOV     AX,FS            ; Data for VRAM high
        MOV     GS:[+16H],AX     ; Write to pix map segment
                                 ; address
; Destination map is 1024-by-768 pixel mode and Intel format
        MOV     AX,1023          ; Value for pix map width
        MOV     GS:[+18H],AX     ; Write to width register
        MOV     AX,767           ; Value for pix map height
        MOV     GS:[+1AH],AX     ; Write to height register
; Bitmap of pixel format register:
; 7 6 5 4 3 2 1 0 <= bits
; | | | | | |_|_|______ pixel image size (* = selected value)
; | | | | |             000 = 1 bit per pixel
; | | | | |             001 = 2 bits per pixel
; | | | | |             010 = 4 bits per pixel
; | | | | |            *011 = 8 bits per pixel
; | | | | |_______ format control
; | | | |           1 = Motorola order
; | | | |          *0 = Intel order
; |_|_|_|______________________________ RESERVED
        MOV     AL,3             ; Select Intel order and 3bbp
                                 ; per pixel
        MOV     GS:[+1CH],AL     ; Write to format register
;**********************|
;    map B is source   |
;    (system memory)   |
;**********************|
        MOV     AL,02            ; Data value for Map B
        MOV     GS:[+12H],AL     ; Write to pixel map index
; AX = offset of source bitmap (in stack)
; DS = segment of source bitmap
; To convert logical address to physical address, the segment
; value is shifted left 4 bits and the offset added
        MOV     EAX,0            ; Clear 32 bits
        MOV     AX,DS            ; Segment to AX
        SHL     EAX,4            ; Shift segment 4 bits
        POP     BP               ; Offset to BP
        ADD     AX,BP            ; Add offset to segment
        MOV     GS:[+14H],EAX    ; Write to pix map base address
```

### 9.6.3 Pattern Map bitBlt

Bitmaps encoded in color depths of 8 bits per pixel or higher take up considerable memory. For example, a 600-by-600 pixel image would require 360,000 bytes of system memory storage. This is more than half of the transient memory space available for MS-DOS applications. Furthermore, any data structure larger than 64K exceeds the capacity of a single segment register and, therefore, creates storage complications for some software. Regarding full-color images there is little that can be done to reduce the storage requirements at execution time. However, it would be a considerable waste to encode in full-color depth the bitmap of an image to be displayed in a single color.

The XGA coprocessor supports the display of monochrome images encoded in a one bit-per-pixel format on a destination of greater color depth. This operation, called a *pattern map* bitBlt, is based on displaying the source bitmap using the current foreground color, as stored in the coprocessors's Foreground Color register at offset 58H.

In this pixBlt mode the foreground source and the background source colors are both selected (code 00) in byte 3 of the Pixel Operations register at offset 7CH. Also the Pattern *x* Address and Pattern *y* Address registers at offset 74H and 76H are used instead of the Source *x* Address and Source *y* Address registers, as is the case in the full color depth pixBlt procedure listed earlier in this section. The following procedure shows the required processing:

```
;****************************************************************
;        processing operations for a pattern map bitBlt
;****************************************************************
P_CODE   SEGMENT PUBLIC
         ASSUME  CS:P_CODE
.386
;
COP_PATBLT        PROC    FAR
; Use graphics coprocessor to perform a pixBlt from a source in
; system memory to a destination in video memory
; Image map is encoded in 1 bit per pixel (pattern map)
; Code assumes:
;        1. 1024-by-768 mode in 256 colors
;        2. The INIT_COP routine has been previously called to
;           initialize the GS and FS segment registers
;        3. Image bitmap is 1 bit per pixel
; On entry:
;          DS:SI = offset of source bitmap in RAM
;          CX = source map pixel width
;          DX = source map pixel height
;          SI = x coordinate of video image
;          DI = y coordinate of video image
;          BL = 8-bit color code to use in displaying image
;*********************|
;   test for not busy |
;*********************|
;                           00 = foreground color
;                          *10 = source pixel map
;                      pppp = function
;                           0010 = draw and step read
;                           0011 = line draw read
;                           0100 = draw and step write
;                           0101 = line draw write
;                          *1000 = pixBlt
;                           1001 = inverting pixBlt
;                           1010 = area fill pixBlt
;                       BYTE 3 = 00001000B = 28H
; byte 2 = SSSS|DDDD (* = values for this operation)
;                      SSSS = source pixel map
;                           0001 = pixel map A
;                          *0010 = pixel map B
;                           0011 = pixel map C
;                      DDDD = destination pixel map
;                          *0001 = pixel map A
;                           0010 = pixel map B
;                           0011 = pixel map C
;                       BYTE 2 = 00100001B = 21H
; byte 1 = PPPP|0000 (* = values for this operation)
;                      PPPP = pattern pixel map
;                           0001 = pixel map A
;                           0010 = pixel map B
;                           0011 = pixel map C
;                          *1000 = foreground (fixed)
;                           1001 = generated from source
;                       BYTE 1 = 00100000B = 80H
; byte 0 = mm00|0oox (* = values for this operation)
;                      mm = mask pixel map
;                          *00 = mask map disabled
;                           01 = boundary enabled
;                           10 = mask map enabled
;                      oox = octant bits (x = don't care)
;                          *00 = start at top left and move
;                                right and down
;                           10 = start at top right and move
;                                left and down
;                           01 = start at bottom left and move
;                                right and up
;                           11 = start at bottom right and move
;                                left and up
;                       BYTE 0 = 00000000B = 00H
        MOV     EAX,028218000H  ; Value from bitmap
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
        RET
COP_SYSVID      ENDP
P_CODE          ENDS
```

```
; AX = offset of source bitmap (in stack)
; ES = segment of source bitmap
; To convert logical address to physical address the segment
; value is shifted left 4 bits and the offset added
        MOV     EAX,0           ; Clear 32 bits
        MOV     AX,ES           ; Segment to AX
        SHL     EAX,4           ; Shift segment 4 bits
        POP     BP              ; Offset to BP
        ADD     AX,BP           ; Add offset to segment
        MOV     GS:[+14H],EAX   ; Write to pix map base address
; Dimensions of source map are in CX and DX registers
        DEC     CX
        DEC     DX
        MOV     GS:[+18H],CX    ; Write to width register
        MOV     GS:[+1AH],DX    ; Write to height register
        MOV     AL,0BH          ; Select Motorola order and
                                ; 8 bits per pixel
        MOV     GS:[+1CH],AL    ; Write to format register
;
;**********************|
;    select mix mode   |
;**********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
;
;**********************|
; coordinates of source|
;   and destination    |
;**********************|
; Source coordinate are 0,0, destination coordinates are in SI
; and DI
        MOV     AX,0            ; Source coordinates
        MOV     GS:[+70H],AX    ; Write to Source x Address
        MOV     GS:[+72H],AX    ; Write to Source y Address
        MOV     GS:[+78H],SI    ; Write to Destination x Address
        MOV     GS:[+7AH],DI    ; Write to Destination y Address
; Store width in Operations Dimension 1 register
        MOV     GS:[+60H],CX    ; Write to Operation Dimension 1
; Store height in Operations Dimension 2 register
        MOV     GS:[+62H],DX    ; Write to Operation Dimension 2
;
;**********************|
;  setup pix operation |
;      registers       |
;**********************|
; Bitmap of Pixel Operations register for pixBlt operation:
; byte 3 = bbss|pppp (* = values for this operation)
;                     bb = background source
;                         *00 = background color
;                          10 = source pixel map
;                     ss = foreground source
```

```
COP_SYSVID      PROC    FAR
; Use graphics coprocessor to perform a pixBlt from a source in
; system memory to a destination in video memory
; Image map is encoded in 8 bits-per-pixel format
; XGA mode is number 2, also in 8-bits-per-pixel
; Code assumes:
;       1. XGA mode 2, 1024-by-768 mode in 256 colors
;       2. The INIT_COP routine has been previously called to
;          initialize the GS and FS segment registers
;       3. Image bitmap is 8 bits per pixel
; On entry:
;         DS:SI = offset of source bitmap in RAM
;         CX = source map pixel width
;         DX = source map pixel height
;         SI = x coordinate of video image
;         DI = y coordinate of video image
;
;**********************|
;   test for not busy  |
;**********************|
        CALL    COP_RDY         ; Local routine
; At this point the coprocessor is not busy
;**********************|
; map A is destination |
;   (video memory)     |
;**********************|
        PUSH    AX              ; Bitmap offset to stack
        MOV     AL,01H          ; Data value for Map A
        MOV     GS:[+12H],AL    ; Write to pixel map index
        MOV     AX,0H           ; Data value for VRAM low
        MOV     GS:[+14H],AX    ; Write to pix map base address
; FS register holds the high-order word of VRAM address. This
; value is calculated by the INIT_COP routine in this module
        MOV     AX,FS           ; Data for VRAM high
        MOV     GS:[+16H],AX    ; Write to pix map segment
                                ; address
; Destination map is 1024-by-768 pixel mode and Intel format
        MOV     AX,1023         ; Value for pix map width
        MOV     GS:[+18H],AX    ; Write to width register
        MOV     AX,767          ; Value for pix map height
        MOV     GS:[+1AH],AX    ; Write to height register
        MOV     AL,3            ; Select Intel order and
                                ; 8 bits per pixel
        MOV     GS:[+1CH],AL    ; Write to format register
;**********************|
;    map B is source   |
;    (system memory)   |
;**********************|
; Note that entry location of map is by the ES segment register
        MOV     AL,02           ; Data value for Map B
        MOV     GS:[+12H],AL    ; Write to pixel map index
```

```
                    CX = 512
                    DX = 384
                    SI = 100
                    DI =  80
                    BL = 00001100B
```

then an 100-by-80 pixel rectangle is drawn with its left-top corner at the center of the screen. If the default palette is active, the color of the rectangle is bright red.

Notice, in the previous example, that the direction octant bits in byte 0 of the Pixel Operations register determine the direction in which the pixBlt takes place. For performing a nonoverlapping pixBlt the direction octant bits are normally set to 0. However, if the source and destination rectangles overlap, the direction octant bits must be used in order to avoid pixel corruption. Table 9.5 shows the action of the direction octant bits in pixBlt operations. These bits are interpreted differently during the coprocessor line draw functions.

Table 9.5 *Direction Octant Bits During PixBlt*

| VALUE | ACTION |
|---|---|
| 00x | From top left to bottom right |
| 10x | From top right to bottom left |
| 01x | From bottom left to top right |
| 11x | From bottom right to top left |
| x = dont't care | |

The procedure named COP_RECT_2 can be used to perform a rectangular fill pixBlt operation.

## 9.6.2 System Memory to Video Memory PixBlts

Another frequent use of the pixBlt operation is to display an image stored in the application's memory space. In this operation the color depth of the source and the destination map usually match; that is, if the video mode is 8 bits-per-pixel, then the image's color depth should also be 8 bits-per-pixel. A mismatch between the color depth of source and destination can give rise to unpredictable errors in execution of the pixBlt, or later in the code. The one exception to the pixel depth match requirement is the use of a 1-bit-per-pixel image, usually called a pattern map, which is described later in this section. In the following procedure a bit-mapped image, encoded in 8-bits-per-pixel format, is displayed while in XGA mode 2:

```
;****************************************************************
;       processing operations for system-to-video bitBlt
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
```

```
;                            0010 = draw and step read
;                            0011 = line draw read
;                            0100 = draw and step write
;                            0101 = line draw write
;                           *1000 = pixBlt
;                            1001 = inverting pixBlt
;                            1010 = area fill pixBlt
;                       BYTE 3 = 00001000B = 08H
; byte 2 = SSSS|DDDD (* = values for this operation)
;                      SSSS = source pixel map
;                           *0001 = pixel map A
;                            0010 = pixel map B
;                            0011 = pixel map C
;                      DDDD = destination pixel map
;                           *0001 = pixel map A
;                            0010 = pixel map B
;                            0011 = pixel map C
;                       BYTE 2 = 00010001B = 11H
; byte 1 = PPPP|0000 (* = values for this operation)
;                      PPPP = pattern pixel map
;                            0001 = pixel map A
;                            0010 = pixel map B
;                            0011 = pixel map C
;                           *1000 = foreground (fixed)
;                            1001 = generated from source
;                       BYTE 1 = 10000000B = 80H
; byte 0 = mm00|0oox (* = values for this operation)
;                      mm = mask pixel map
;                           *00 = mask map disabled
;                            01 = boundary enabled
;                            10 = mask map enabled
;                      oox = octant bits (x = don't care)
;                           *00 = start at top left and move
;                                 right and down
;                            10 = start at top right and move
;                                 left and down
;                            01 = start at bottom left and move
;                                 right and up
;                            11 = start at bottom right and move
;                                 left and up
;                       BYTE 0 = 00000000B = 00H
        MOV     EAX,08118000H   ; Value from bitmap
        MOV     GS:[+7CH],EAX   ; Write to Pixel Operations
                                ; register
        RET
COP_RECT_2      ENDP
P_CODE          ENDS
```

If XGA is initialized to 1024-by-768 pixels in 256 colors, and if on entry to the procedure COP_RECT_2 (listed above) the CPU is initialized as follows:

```
;**********************|
;   prepare to pixBlt  |
;**********************|
        MOV     AL,01H          ; Data value for Map A
        MOV     GS:[+12H],AL    ; Write to pixel map index
        MOV     AX,0H           ; Data value for VRAM low
        MOV     GS:[+14H],AX    ; Write to pix map base address
; FS register holds the high-order word of VRAM address. This
; value is calculated by the INIT_COP routine in this module
        MOV     AX,FS           ; Data for VRAM high
        MOV     GS:[+16H],AX    ; Write to pix map segment
                                ; address
; Code assumes 1024-by-768 pixel mode and Intel format
        MOV     AX,1023         ; Value for pix map width
        MOV     GS:[+18H],AX    ; Write to width register
        MOV     AX,767          ; Value for pix map height
        MOV     GS:[+1AH],AX    ; Write to height register
        MOV     AL,3            ; Select Intel order and 8 bits
                                ; per pixel
        MOV     GS:[+1CH],AL    ; Write to format register
;**********************|
;    perform pixBlt    |
;**********************|
        MOV     AL,03H          ; Select source mix mode
        MOV     GS:[+48H],AL    ; Write to Mix register
; Write color (in BL) to Foreground color register
        MOV     GS:[+58H],BL    ; Write to Foreground color
                                ; register
; Write coordinates of rectangle's start point to coprocessor
; registers
        MOV     GS:[+78H],CX    ; Write to Destination x Address
                                ; register
        MOV     GS:[+7AH],DX    ; Write to Destination y Address
                                ; register
; Store width in Operations Dimension 1 register
        MOV     GS:[+60H],SI    ; Write to Operation Dimension 1
; Store height in Operations Dimension 2 register
        MOV     GS:[+62H],DI    ; Write to Operation Dimension 2
;**********************|
;  setup pix operation |
;      registers       |
;**********************|
; Bitmap of Pixel Operations register for pixBlt operation:
; byte 3 = bbss|pppp (* = values for this operation)
;                     bb = background source
;                         *00 = fixed register pixBlt
;                          10 = VRAM to VRAM pixBlt
;                     ss = foreground source
;                         *00 = fixed register pixBlt
;                          10 = VRAM to VRAM pixBlt
;                     pppp = function
```

The action performed by each field of the Pixel Operations register is explained in the discussion of the various coprocessor commands contained in the sections that follow.

## 9.6 XGA PixBlt Operations

A pixel block transfer operation (called a pixBlt in XGA documentation) consists of moving rectangular memory blocks from a source area to a destination area. Both the source and the destination can be system or video memory. The dimensions of the pixel rectangles are entered into the Operations Dimension registers: the width into Operations Dimension 1 and the height into Operations Dimension 2. The pixBlt can be programmed to start at any one of the four corners of the rectangle. The operation always proceeds in the direction of the diagonally opposite corner. The direction is entered into the Pixel Operations register at offset 7CH.

### 9.6.1 Rectangular Fill pixBlt

Perhaps the simplest pixBlt operation is filling a rectangular screen area using the Foreground Color register as source data. The following procedure shows the coprocessor commands necessary to perform this form of pixBlt:

```
;****************************************************************
;       processing operations for rectangular pixBlt
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
;
COP_RECT_2      PROC    FAR
; Use graphics coprocessor to perform a pixBlt operation
; on a rectangular screen area
; Code assumes:
;       1. 1024-by-768 mode in 256 colors (mode number 2)
;       2. The INIT_COP routine has been previously called to
;          initialize the GS and FS segment registers
; On entry:
;         CX = x coordinate of top-left corner
;         DX = y coordinate of top-left corner
;         SI = width of rectangle, in pixels
;         DI = height of rectangle, in pixels
;         BL = 8-bit color value according to current palette
;
;*********************|
;   test for not busy  |
;*********************|
        CALL    COP_RDY         ; Local routine
; At this point the coprocessor is not busy
```

Table 9.4 *Coprocessor Logical and Arithmetic Mixes*

| CODE | HEX | ACTION |
|---|---|---|
| 0 | 00H | Zeros |
| 1 | 01H | Source AND destination |
| 2 | 02H | Source AND NOT destination |
| 3 | 03H | Source |
| 4 | 04H | NOT source AND destination |
| 5 | 05H | Destination |
| 6 | 06H | Source XOR destination |
| 7 | 07H | Source OR destination |
| 8 | 08H | NOT source AND NOT destination |
| 9 | 09H | Source XOR NOT destination |
| 10 | 0AH | NOT destination |
| 11 | 0BH | Source OR NOT destination |
| 12 | 0CH | Source NOT destination |
| 13 | 0DH | NOT source OR destination |
| 14 | 0EH | NOT source OR NOT destination |
| 15 | 0FH | Ones |
| 16 | 10H | Maximum |
| 17 | 11H | Minimum |
| 18 | 12H | Add with saturate |
| 19 | 13H | Destination minus source (with saturate) |
| 20 | 14H | Source minus destination (with saturate) |
| 21 | 15H | Average |
| 22 | 16H | Reserved |
| . | . | |
| . | . | |
| 255 | FFH | |

Direction Steps register (at offset 2CH). The Pixel Operations register also defines the flow of data during coprocessor operations. Figure 9.9 is a bitmap of the Pixel Operations register.

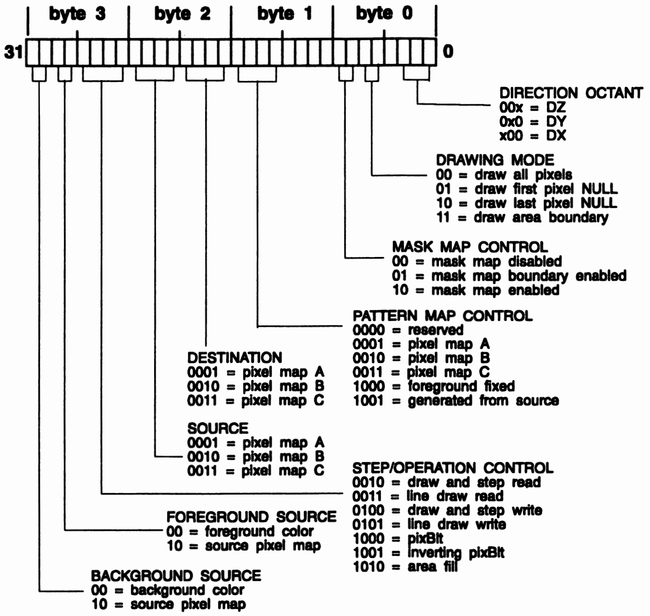


Figure 9.9 *Pixel Operations Register Bitmap*

## 9.5.5 Pixel Masking and Color Compare Operations

In addition, it is possible to protect individual pixels by masking. The Pixel Bit Mask register (offset 50H) is used for this purpose. A value of 1 in the Pixel Bit Mask register enables the corresponding pixel for update, while a value of 0 determines that the pixel is excluded from the update operation. Note that the Pixel Bit Mask is related to the adopted format. In 8-bits-per-pixel mode, the Pixel Bit Mask has active the eight low-order bits of the register, while in a 2 bit-per-pixel mode only the lowest two bits are used.

The coprocessor also allows a color compare operation that further inhibits certain pixel patterns from upgrade. The Destination Color Compare Value register (offset 4CH) is used for storing the bitmap to be used in the comparison. As with the Pixel Bit Map register, the number of bits effectively used in the color compare operation depends on the number of bits per pixel in the adopted format. Several color compare conditions are allowed. The code for the selected condition is stored in the Destination Color Compare Condition register (offset 4AH). Table 9.3 lists the condition codes.

Table 9.3 *Destination Color Compare Conditions*

| CODE | BINARY | CONDITION |
|---|---|---|
| 0 | 000 | Always true (disable update) |
| 1 | 001 | Destination = color compare value |
| 2 | 010 | Destination = color compare value |
| 3 | 011 | Destination = color compare value |
| 4 | 100 | Always false (enable update) |
| 5 | 101 | Destination = color compare value |
| 6 | 110 | Destination = color compare value |
| 7 | 111 | Destination = color compare value |

## 9.5.6 Mixes

In Figure 9.8 we see that the attribute of the destination pixels depends upon a mix. The mix is a logical or arithmetic operation used in combining the source and the destination bitmaps. The mix is selected independently for the foreground and the background pixels. (See Figure 9.8.) The foreground mix is entered into the Foreground Mix register (offset 48H) and the background mix into the Background Mix register (offset 49H). The actual mix operation is determined by a mix code. The mix codes and actions are shown in Table 9.4.

The word *saturate* in Table 9.4 means that if the result of an addition or subtraction operation is greater than 1, the final result is left at 1, while if it is smaller than 0, it is left at 0.

## 9.5.7 Pixel Operations

The coprocessor starts executing the programmed operation when data is written to the Pixel Operations register (offset 7CH). The one exception to this statement is the draw-and-step command which is initiated by writing to the

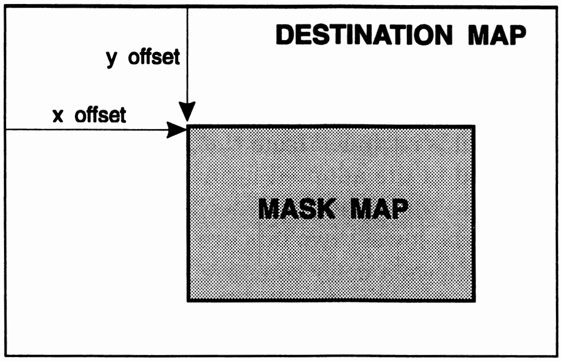


Figure 9.7 *Mask Map x and y Offset*

### 9.5.4 Pixel Attributes

The coprocessor generates a pixel with specific attributes by combining the source, pattern, and destination according to a certain mix mode. The pattern pixel map, if used, serves as a filter to determine if a bit corresponds to a foreground or a background pixel. A value of 1 in the pattern pixel map determines that the bit is mapped to a foreground pixel; a value of 0 determines that the bit is mapped to a background pixel. If no pattern map is used, then the foreground and background sources can be a specific color or determined by the color encoding stored in a source map. If the foreground source is a specific color, it is stored at the Foreground Color register at offset 58H. The background color is stored at the register at offset 5CH. The elements that take part in determining a pixel's attributes are shown in Figure 9.8.

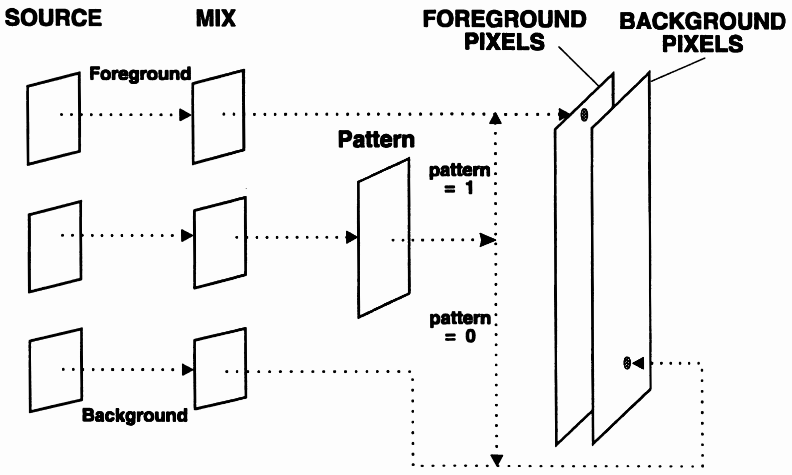


Figure 9.8 *Determining the Pixel Attribute*

of video memory is entered in the Pixel Map *n* Base Pointer register and the actual position within the video display is determined by the *x* and *y* coordinates entered in the Destination Map *x* Coordinate and Destination Map *y* Coordinate registers. On the other hand, if the pixel map is within the application's address space, the offset is usually 0. This value signals the start of the pixel map as the reference position; however, the coordinates can be changed to indicate another position within the defined rectangle.

The masking mode is selected by a two-bit field in the Pixel Operations register. The difference between the Mask Map Enabled and the Boundary Enabled modes can be seen in Figure 9.6.

Coordinate registers for source and pattern pixel maps are located at offset 70H and 74H. (See Table 9.2.) However, there are no *x* and *y* coordinate registers for the mask map, because its origin is assumed to coincide with that of the destination map. Nevertheless, if the mask map is smaller than the destination map, it becomes necessary to locate the mask map within the destination map. This is done by means of the Mask Map Origin *x* Offset and the Mask Map Origin *y* Offset registers at offset 6CH and 6EH, respectively. The use of these mask map offset values is shown in Figure 9.7.

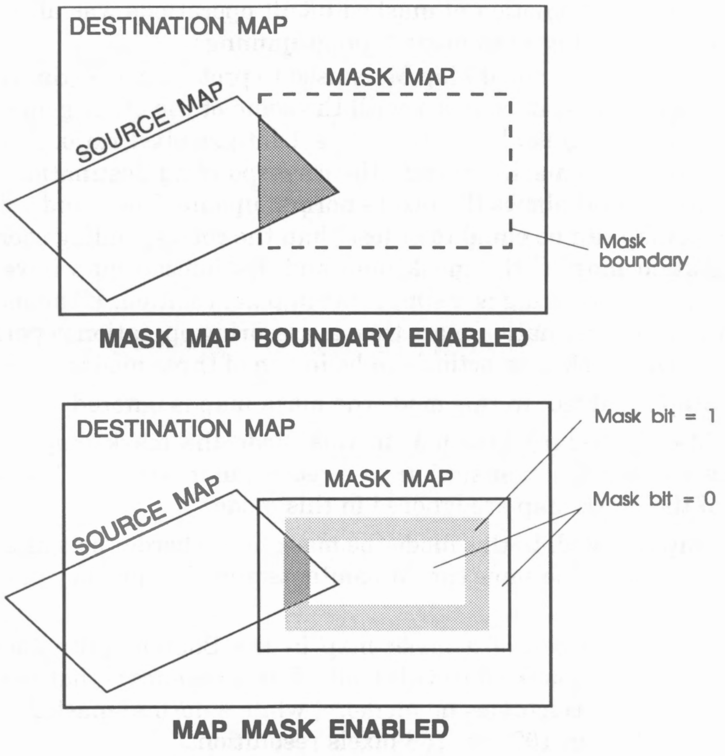


Figure 9.6 *XGA Mask Map Operations*

2. In an operation that consists of reading video data into system memory the source is a video memory map and the destination a location in the application's memory space.
3. An operation that copies a video image into another screen area has both source and destination in video memory.
4. The coprocessor can also copy an area of user memory into another one. In this case both source and destination maps are located in the application's memory space.
5. A pattern map is used to encode a 1-bit-per-pixel image. In this case a value of 1 indicates a foreground pixel and a value of 0 a background pixel. In this type of pixBlt the pattern map is the source map and the destination map is in video memory.

Several forms of pixBlt operations using the general-purpose maps A, B, and C are illustrated in the code samples listed later in this section.

## 9.5.3 The Mask Map

The mask map is an additional type of pixel map closely related to the destination map. The notion of masked bitBlt operations was also mentioned in Chapter 7 regarding VGA mode X programming.

The mask map, also called Map M, is used to protect the destination map on a pixel-by-pixel basis. In contrast with the some of the other general-purpose maps, the mask map is always fixed to a 1 bit-per-pixel ratio. A 0 bit in the mask map (inactive mask) protects the corresponding destination pixel from update, while a 1 bit allows the pixel's normal update. The *x* and *y* dimensions of the mask map can be equal to or less than the corresponding coordinates in the destination map. If the mask map and destination map have the same dimensions, then masking is a simple bit-to-pixel relation. If the mask map is smaller than the destination map, then a scissoring operation is performed. In this respect the mask map action can be in one of three modes, as follows:

1. Mask Map Disabled. In this mode the mask map is ignored.
2. Mask Map Boundary Enabled. In this mode the mask map performs an outline scissoring action similar to a rectangular window. The actual contents of the mask map are ignored in this mode.
3. Mask Map Enabled. In this mode the mask map's border acts as a scissoring rectangle, and at the same time its contents provide a pixel by pixel masking operation.

Notice that the action of a mask map in the Boundary Enabled mode is identical to that of a mask map of all 1 bits. The difference is that the Boundary Enabled mask map consumes no memory, while a normal mask map can take up as much as 94K in 1024-by-768 pixels resolution.

In addition to the map address, the program can define the pixel map's *x* and *y* coordinates. These value can be interpreted as offsets within the map. For example, if the destination pixel map is the video screen, the physical address

1. The Pixel Map *n* Base Pointer register (at offset 14H) contains the map's start address.
2. The Pixel Map *n* Width register (at offset 18H) determines the horizontal dimension of the pixel map and the Pixel Map *n* Height register (at offset 1AH) determines its vertical dimension. The values loaded into these registers must be one less than the required size.
3. The Pixel Map Format register (at offset 1CH) determines if the map is in 1, 2, 4, or 8 bits per pixel in the original XGA and up to 16 bits per pixel in XGA-2, and also whether it is encoded in Intel or Motorola data format.
4. The Pixel Map Index register (at offset 12H) is used to determine if the mask map is of type A, B, C, or M. The different mask map types are explained later in this section.

The $x$ and $y$ coordinates of a pixel map are based on the same convention used for the video display; that is, the top-left corner of the pixel map has coordinates $x = 0, y = 0$. The value of $x$ increases to the right and the value of $y$ increases downward. The pixel map coordinate system conventions and dimensions are shown in Figure 9.5.

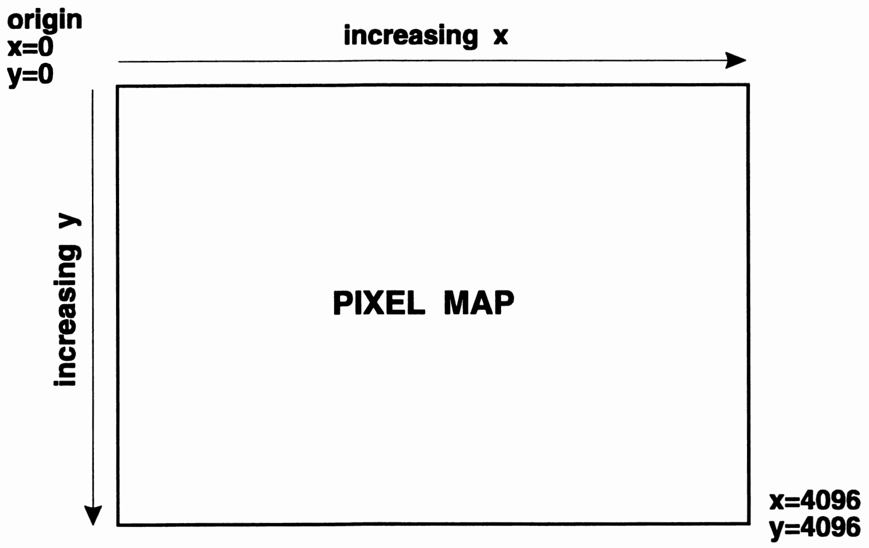


Figure 9.5 *XGA Pixel Map Coordinates*

In relation to the coprocessor operation a pixel map can represent a source, a destination, or a pattern. The following cases illustrate common bitBlt operations:

1. In displaying a bitmap stored in the applications address space the source map is the application's data, and the destination map is a location in video memory.

bit is set, then the code can proceed with the next coprocessor operation. At this time the code must write a one to bit number 7 in order to clear the interrupt condition so that the next interrupt can take place.

Since polling the busy bit is easier to implement in software, this is the method illustrated in this section. The main objection to polling for hardware not busy is that it slows down operations since the coprocessor must delay execution to read its own Control register. This can be partially overcome by designing routines that include a delay loop so that the coprocessor is not polled constantly. The following procedure polls bit 7 of the coprocessor Control register to test for a not-busy condition. The COP_RDY procedure is usually called by other graphics primitives before emitting a new coprocessor command. The delay period in the wait loop is an arbitrary value.

```
COP_RDY          PROC    NEAR
; Poll bit 7 of coprocessor Control register (offset 11H) to
; determine if coprocessor is busy, if so, wait until ready
; Code assumes that GS segment holds coprocessor base address
;
        PUSH    AX                ; Save context
        PUSH    CX
TEST_COP:
        MOV     AL,GS:[+11H]      ; Read Control register
        TEST    AL,10000000B      ; Test bit 7
        JZ      COP_READY         ; Go if bit is clear
        MOV     CX,100            ; Counter for wait loop
; A 100-iteration wait loop is introduced so that the coprocessor
; is not polled constantly, thus slowing down execution
WAIT_100:
        NOP                       ; Delay
        NOP
        LOOP    WAIT_100          ; Wait
        JMP     TEST_COP          ; Test again after wait
COP_READY:
        POP     CX                ; Restore context
        POP     AX
        RET
COP_RDY         ENDP
```

## 9.5.2 General-Purpose Maps A, B, and C

The XGA graphics coprocessor can operate on three general-purpose pixel maps, designated as Map A, Map B, and Map C in the IBM literature. The identification letters A, B, and C are sometimes generically represented by the variable *n*, as is the case in the Pixel Map *n* Base Pointer designation used in Table 9.2. Note that, in actual coding, Map *n* is either Map A, Map B, or Map C. Pixel maps can be located in system or in video memory. The maximum size of a map is 4096-by-4096 pixels.

The following coprocessor registers are related to pixel maps:

At this point the coprocessor is ready for use. The procedure INIT_COP in the XGA4 module of the VIDEO.LIB uses similar processing to initialize the coprocessor. The programmer must consider that, if this initialization method is used, the software must make sure that the 80386 segment registers FS and GS are preserved, since their contents are repeatedly required in setting up and performing coprocessor operations.

## 9.5 Programming Coprocessor Operations

The XGA graphics coprocessor can execute drawing operations in parallel with the CPU. The original XGA coprocessor can execute in 1-, 2-, 4-, and 8-bits-per-pixel formats, but not in the direct color mode. However, in XGA-2 the coprocessor is also available in the direct color mode. The execution of a coprocessor operation requires the following steps:

1. The CPU initializes the coprocessor registers to be used in the operation.
2. Coprocessor operation starts when the CPU writes a command to the Pixel Operations register.
3. The coprocessor executes the programmed operation. During this time the system microprocessor can be performing other tasks.

The graphics functions that can be performed by the coprocessor are pixel block transfer (abbreviated pixBlt), line draw, and draw-and-step.

The programmer can set up the coprocessor so that it generates an interrupt at the conclusion of its operations. This mechanism can be used in optimizing parallel processing, in task switching in a multitasking environment, in error recovery, in figure animation, and in synchronizing coprocessor access. The coprocessor Operation Complete interrupt is enabled by setting bit 7 of the Interrupt Enable register of the Display Controller group. The interrupt source is identified by testing the corresponding bit in the Interrupt Status register of the Display Controller group. Note that this bit is set if an interrupt occurred, regardless of the setting of the Interrupt Enable register.

### 9.5.1 Synchronizing Coprocessor Access

Since the coprocessor operates asynchronously regarding the CPU, the central processor must wait until the coprocessor has concluded its previous operation before issuing a new command. This can be performed in two ways: by enabling the Coprocessor Operation Complete interrupt described in the previous paragraph or by polling the busy bit in the coprocessor Control register. Both methods are quite feasible, each having its advantages and disadvantages.

An XGA interrupt handler for testing the conclusion of coprocessor operation (or any other XGA interrupt for that matter) is designed to intercept vector 0AH, which corresponds to the IRQ2 line of the system's Interrupt Controller. Since this interrupt can be shared, the handler must first make sure that the interrupt was caused by the coprocessor. This requires testing bit 7 of the Interrupt Status register (at offset 05H). If the Coprocessor Operation Complete

```
; Mode register of the XGA Display Controller (offset + 9)
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,9            ; To Mode register
        MOV     AL,03H          ; 7 6 5 4 3 2 1 0  <= bitmap
                                ; | | | | | | | |   Bits/pixel
                                ; | | | | | |_|_|__ 000 = 1 bit
                                ; | | | | |         001 = 2 bits
                                ; | | | | |         010 = 4 bits
                                ; | | | | |        *011 = 8 bits
                                ; | | | | |         100 = 16 bits
                                ; | | | | |___  *0 = Intel
                                ; | | | |        1 = Motorola
                                ; |_|_|_|____ RESERVED
                                ; 03H = 00000011B
        OUT     DX,AL
; ... continues in the following code listing
```

### 9.4.4 Initializing Coprocessor Registers

Some coprocessor registers should be initialized for normal operation. This includes the Coprocessor Control register at offset 11H, as well as the Destination Color Compare, the Plane Mask, and the Carry Chain Mask registers.

```
; ... continues from the previous code listing
;**********************|
;  init cop. registers |
;**********************|
; Some coprocessor registers must be initialized for normal
; operation
; 1. Reset coprocessor control register
        MOV     AL,0H           ; Data value for Coprocessor
                                ; Control
        MOV     GS:[+11H],AL    ; Write to register
; 2. Set Destination Color Compare register to always false to
; enable updates
        MOV     AL,4H           ; Data value for Color Compare
        MOV     GS:[+4AH],AL    ; Write to register
; 3. Set Plane Mask register for updating all planes
        MOV     AL,0FFH         ; Data value for Plane Mask
        MOV     GS:[+50H],AL    ; Write to register
; 4. Set the Carry Chain Mask register for 8 bpp propagation
        MOV     AL,0FFH         ; Data value for Carry Chain Mask
        MOV     GS:[+54H],AL    ; Write to register
;**********************|
;   restore and exit   |
;**********************|
        POP     DS              ; Restore caller's DS
        RET
INIT_COP        ENDP
P_CODE          ENDS
```

The required processing for calculating the VRAM physical address is shown in the following code fragment:

```
; ... continues from the previous code listing
;*********************|
;    get VRAM base    |
;*********************|
; Start of video memory is a 32-bit physical address determined
; as follows:
;            high-order word      low-order word
;
;          ___________________  ____________________
;          BBBB BBBi ii00 0000 | 0000 0000 0000 0000
;
; B (base) = are 7 high bits in POS register 4
; i (instance) = bits 1 to 3 in POS register 2
; The high-order word of the video memory address must be stored
; at the coprocessor register at offset 16H during operations
        MOV     AL,POS_4        ; Get B bits in above map
        AND     AL,11111110B    ; Clear low bit
        SHL     AX,8            ; Shift to high position
; Get instance bits
        MOV     BL,POS_2        ; Get POS register 2
        AND     BL,00001110B    ; Mask out other bits
        MOV     BH,0            ; Clear high part of BX
        SHL     BX,5            ; Move instance bits to position
        OR      AX,BX           ; OR them with B bits (in AX)
        MOV     FS,AX           ; Store in FS segment
; ... continues in the following code listing
```

Notice that the high-order part (16 bits) of the VRAM physical address is now stored in the FS segment register. The 80386 FS segment is a convenient storage for this value, which later is used in coprocessor programming.

## 9.4.3 Selecting the Access Mode

Coprocessor operation requires that the Memory Access Mode register of the Display Controller be set to 1, 2, 4, or 8 bits per pixel in original XGA and up to 16 bits per pixel in XGA-2. The data storage format also must be selected. The options are the Intel (little-endian) or Motorola (big-endian) formats. In the PC environment with a fully equipped XGA (1Mb of VRAM) the coprocessor is typically set to 8 bits per pixel to match the Intel format of the CPU. The following code fragment shows selecting the access mode for coprocessor operation:

```
; ... continues from the previous code listing
;*********************|
;  select access mode |
;*********************|
; Select Intel order and 8 bits per pixel in the Memory Access
```

```
        MOV     AL,POS_2        ; Get POS register 2
        AND     EAX,0F0H        ; Preserve ROM bits
        SHR     EAX,4           ; Shift ROM to low nibble
        MOV     ECX,2000H       ; Multiplier
        MUL     ECX             ; EAX * ECX in EAX
        ADD     EAX,0C0000H     ; Add constant
        MOV     EBX,EAX         ; Store ROM address in EBX
; EBX now holds ROM address
; Instance is stored in bits 1 to 3 of POS register 2
        MOV     EAX,0           ; Clear EAX
        MOV     AL,POS_2        ; Get POS register 2
        AND     EAX,0EH         ; Preserve instance bits
        SHR     EAX,1           ; Shift right instance bits
        MOV     ECX,128         ; Multiplier to ECX
        MUL     ECX
        ADD     EAX,1C00H       ; Add constant from formula
; Add ROM address
        ADD     EAX,EBX
        SHR     EAX,4           ; Shift right one nibble to
                                ; to obtain segment value
; Store segment value in GS
        MOV     GS,AX           ; Move segment into GS
; ... continues in the following code listing
```

Note that the segment value of the coprocessor base address has been stored in segment register GS. This is consistent with the notion of making full use of the 80386 architecture and instruction set.

## 9.4.2 Obtaining the Video Memory Address

The physical address of video memory is a 32-bit value determined from the video memory base address field in POS register 4 and from the instance field in POS register 2. (See Figure 8.6.) The address is formed by relocating the POS data items as shown in Figure 9.4.

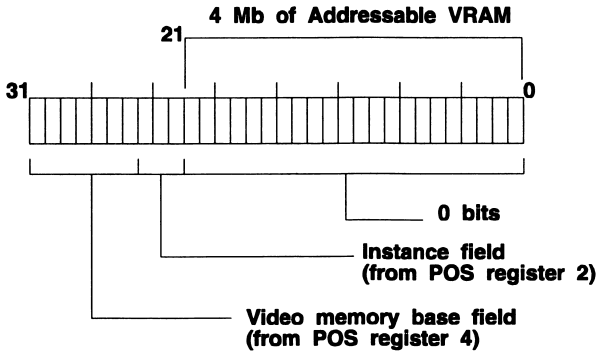


Figure 9.4 *Video Memory Address Bitmap*

```
coprocessor address = (((i * 128) +1C00H) + (R + 2000H) + C000H)
```

**where *i* is the instance and *R* is the value in the ROM field of POS register 2. The following procedure assumes that POS registers 2 and 4 have been read and stored in variables during XGA initialization:**

```
;****************************************************************
;       data variables for coprocessor initialization
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_REG_BASE    DW      0       ; Register base for XGA system
; The following variables are loaded from the XGA POS registers
POS_2           DW      0       ; POS register 2
POS_4           DW      0       ; POS register 4
;
XGA_DIRECT      ENDS
;****************************************************************
;     processing operations for coprocessor initialization
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
;
INIT_COP        PROC    FAR
; Initialize XGA coprocessor
; Code assumes that the procedure INIT_XGA has been called and
; that the POS_x variables have been loaded
;
; Coprocessor base address is calculated as follows:
; ROM address = (ROM field + 2000H) + C0000H
; COP address = (((Instance * 128) + 1C00H) + ROM address)
;
; On exit:
;         GS = coprocessor base address
;         FS = base address of video memory
;         Display controller set for 8 bits-per-pixel in Intel
;         data format. Coprocessor registers initialized for
;         normal operation
;
;*********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT           ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:XGA_DIRECT
; First calculate ROM address from data in POS register 2
        MOV     EAX,0           ; Clear EAX
```

The coprocessor registers can be accessed using either the Intel or the Motorola data formats. Table 9.2 represents the register structure in the Intel format, which is the one most likely to be used in the PC environment. Most coprocessor registers are write-only. The second column in Table 9.2 shows which registers can be read by the CPU. Note that the Current Virtual Address, State A Length, and State B Length registers are read-only. Software should not write to these registers. The Page Directory Base Address and the Current Virtual Address registers (offset plus 0 and plus 4, respectively) are used only in a virtual memory environment. Real mode programs, such as those executing in MS-DOS, need not access these registers.

The XGA coprocessor can access all memory in the system and treats video memory and system memory in the same fashion. Once the coprocessor is informed of the VRAM address, it is used to determine if the memory access is local or remote. In remote accesses the coprocessor obtains direct control of the bus. This capability of the coprocessor improves XGA performance by allowing the CPU to continue executing code while the coprocessor manipulates memory data.

The XGA graphics coprocessor is designed to take advantage of the 80386 instruction set. Since all present implementations of XGA require an 80386 CPU, XGA programs can safely use 80386 instructions without fear of hardware incompatibility. Therefore, in the code samples that follow we have used the 80386 instruction set when programming coprocessor operations.

## 9.4 Initializing the Coprocessor

The first action to be taken by a program that intends to access the XGA coprocessor is its initialization. The fundamental steps of this initialization consist of calculating and storing two data items required in programming this device: the base address of the coprocessor register space and the physical address of the start of video memory. Note that the video memory address used by the coprocessor corresponds to the 4Mb aperture previously mentioned. The data for calculating these addresses is found in the XGA POS registers (see Figure 8.6). The coprocessor initialization routine performs the following operations:

1. Obtains and stores the address of the coprocessor register base.
2. Obtains and stores the address of physical video memory.
3. Selects and initializes the color depth and access mode.
4. Initializes coprocessor registers for normal operation.

### 9.4.1 Obtaining the Coprocessor Base Address

The coprocessor base address is calculated from the ROM address field in POS register 2 and from the instance field in this same POS register. (See Figure 8.6.) The coprocessor address formula is

## 9.3 XGA Graphics Coprocessor Architecture

The present discussion relates to fundamental programming operations on the XGA coprocessor. To the programmer, this chip appears as a set of memory-mapped registers. The area of memory devoted to these registers is called the *coprocessor's address space*. Table 9.2 is a map of the coprocessor registers.

Table 9.2 *XGA Graphic Coprocessor Register Map*

| OFFSET | READ/ WRITE | + 0 | +1 | + 2 | + 3 |
|---|---|---|---|---|---|
| 0 | W | Page Directory Base Address | | | |
| 4 | R | Current Virtual Address | | | |
| 8 | | | | | |
| C | R | State A Length | State B Length | | |
| 10 | R/W<br>W | | Coprocessor Control | Pixel Map Index | |
| 14 | W | Pixel Map n Base Pointer | | | |
| 18 | W | Pixel Map n Width | | Pixel Map n Height | |
| 1C | W | Pixel Map format | | | |
| 20 | R/W | Bresenham Error Term | | | |
| 24 | W | Bresenham K1 Term | | | |
| 28 | W | Bresenham K2 Term | | | |
| 2C<br>.<br>.<br>44 | W | Direction Steps | | | |
| 48 | W | Foreground Mix | Background Mix | Destination Color Compare Condition | |
| 4C | W | Destination Color Compare Value | | | |
| 50 | W | Pixel Bit Mask | | | |
| 54 | W | Carry Chain Mask | | | |
| 58 | W | Foreground Color | | | |
| 5C | W | Background Color | | | |
| 60 | W | Operations Dimension 1 | | Operations Dimension 2 | |
| 64 | | | | | |
| 68 | | | | | |
| 6C | W | map mask Origin x Offset | | map mask Origin y Offset | |
| 70 | R/W | Source Map x Coordinate | | Source Map y Coordinate | |
| 74 | R/W | Pattern Map x Coordinate | | Pattern Map y Coordinate | |
| 78 | R/W | Destination Map x Coordinate | | Destination Map y Coordinate | |
| 7C | W | Pixel Operations | | | |

The actual conversion consists of shifting the 8-bit color fields to the corresponding position of the 16-bit color fields. For example, the most significant red bit in the 8-bit format (bit number 7) is shifted to the most significant red bit position in the 16-bit format (bit number 15). This operation requires an 8-bit left shift. By the same token, the green bits must be shifted by five bit positions and the blue bits by three bit positions. In addition, it is convenient to perform an additional adjustment so that the color values are located at the high end of the allotted range. In this manner, the highest red value in the 8-bit format (11 binary) is converted to the highest color value in the 16-bit format (11111 binary) by ORing with the binary mask 00111. The following code fragment shows the conversion of a color value from an 8-bit to 16-bit format:

```
; Expand a color in RR GGGG BB format to RRRRR GGGGGG BBBBB
; AL holds 8-bit color
; Push registers required for expansion
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,0            ; Clear high nibble
        MOV     DX,AX           ; Copy color in DX
;***********************|
;  expand color fields  |
;***********************|
; Mask off entry color fields
        AND     AL,00000011B    ; Leave blue bits
        SHL     AL,3            ; Multiply by 8
        OR      AX,7H           ; Add minimum range
        MOV     CX,AX           ; Store blue in CX
; Now expand green field
        MOV     AX,DX           ; Reload original color code
        AND     AL,00111100B    ; Leave green bits
        SHL     AX,5            ; Shift to green field
        OR      AX,00E0H        ; Add minimum range
        MOV     BX,AX           ; Store in BX
; Expand red field
        MOV     AX,DX           ; Reload original color code
        AND     AX,11000000B    ; Leave red bits
        SHL     AX,8            ; Shift to red field
        OR      AX,3800H        ; Add minimum range
; OR all color values
        OR      AX,CX           ; OR in blue
        OR      AX,BX           ; and green
; AX now has expanded color in 16-bit RRRRR GGGGGG BBBBB format
        POP     DX              ; Restore registers
        POP     CX
        POP     BX
```

Notice that the conversion always is an approximate value since there can be no exact equivalent between the 8-bit and the 16-bit color values.

```
        MOV     ES,AX           ; To ES segment
; Get address in XGA system
        CLC                     ; Clear carry flag
        MOV     AX,1280         ; This many bytes per line
        MUL     DX              ; DX holds line count of address
        ADD     CX,CX           ; Double the x length
        ADD     AX,CX           ; and add in
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Save offset in BX
        MOV     AX,DX           ; Move bank number to AL
;**********************|
;    change banks      |
;**********************|
        MOV    DX,XGA_REG_BASE  ; XGA Base register address
        ADD    DX,08H           ; Aperture Index register
        OUT    DX,AL            ; Bank number is in AL
        POP     AX              ; Restore color value
;**********************|
;   set the pixel      |
;**********************|
        MOV     ES:[BX],AX      ; Write the dot
        POP     DS              ; Restore caller's DS
        RET
XGA_PIXEL_5     ENDP
P_CODE          ENDS
```

### 9.2.3 16-Bit Color Adjustments

Graphics software often needs to convert a color value from one format to another. For example, a routine or program designed for a 256-color mode (8 bits per pixel) has to be ported to a VGA or XGA direct color mode (16 bits per pixel.) In the simplest variation of this conversion each color value coded in RR-GGGG-BB format needs to be transformed to a color in RRRRR-GGGGGG-BBBBB format. Figure 9.3 shows the color field mapping in both formats.

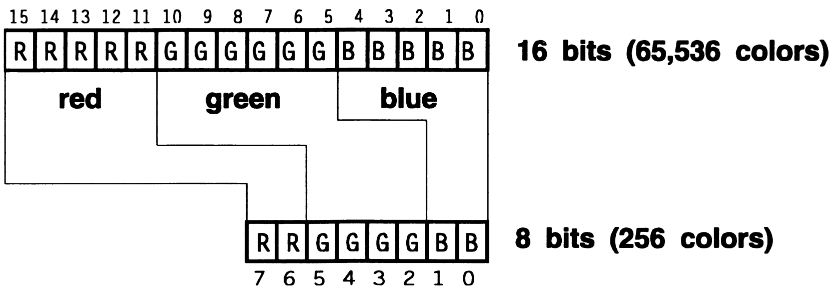


Figure 9.3 *Conversion Map for 8- to 16-Bit Color Modes*

In the XGA direct color mode the actual setting of a screen pixel is performed with a word-write operation, as shown in the following code fragment:

```
; Word write operation for 16 bit-per-pixel mode
; AX = 16-bit color code in 5-6-5 format
; BX = offset into video buffer
; ES = video memory segment (A000H or B000H)
;
        MOV         ES:[BX],AX          ; Writes the pixel
```

In this mode the programmer must take into account that each screen pixel is mapped to two video buffer bytes. For example, the tenth pixel from the start of the first screen row is located 20 bytes from the start of the buffer. By the same token, each pixel is at a word boundary in the video buffer. The display routine must make the necessary adjustment, as in the following procedure:

```
;****************************************************************
;       data variables for XGA direct color pixel set
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_REG_BASE    DW      0       ; Register base for XGA system
;
XGA_DIRECT      ENDS
;****************************************************************
;     processing operations for XGA direct color pixel set
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
.386
XGA_PIXEL_5     PROC  FAR
; Write a screen pixel accessing XGA memory directly in direct
; color mode (mode number 5)
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
;       AX = pixel color in 16-bit RRRRRGGGGGGBBBBB format
; Note: code assumes that XGA is in a 640-by-480 pixel mode
;       in 65,536 colors (mode number 5)
;*********************|
;   save caller's DS  |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        PUSH    AX              ; Save color code
        MOV     AX,XGA_DIRECT   ; Local data segment
        MOV     DS,AX           ; to DS
        ASSUME  DS:XGA_DIRECT
; Set ES to video buffer base address
        MOV     AX,0A000H       ; Base for all graphics modes
```

```
; This value is ANDed with display memory. Setting all bits
; makes the palette visible again
        MOV     AX,0FF64H       ; All bits set
        OUT     DX,AX           ; To make visible
        POP     DS              ; Return caller's DS
        RET
DC_PALETTE      ENDP
;****************************************************************
LOAD_128        PROC    NEAR
; Auxiliary procedure for DC_PALETTE to load a group of 128
; DAC registers with the recommended values
;
        MOV     DX,XGA_REG_BASE ; Base address
        ADD     DX,0AH          ; Index register
        MOV     AX,0065H        ; Select Data register
        OUT     DX,AL
        INC     DX              ; To Data register
        MOV     BX,0            ; BX is value for blue register
        MOV     CX,128          ; Counter for 128 registers
; Loop to send 3 bytes to 128 registers
DC_128:
        MOV     AL,0            ; Send red
        OUT     DX,AL           ; Send to port
        JMP     SHORT $ + 2     ; I/O delay
        OUT     DX,AL           ; Send green
        MOV     AL,BL           ; Load blue value
        OUT     DX,AL           ; Send blue
        ADD     BL,8            ; Bump blue value in BL
                                ; Wraps around automatically
        LOOP    DC_128
        DEC     DX              ; Back to Index register
        RET
LOAD_128        ENDP
P_CODE          ENDS
```

## 9.2.2 Pixel Setting in Direct Color Mode

The programmer working in the direct color mode has fewer options than in other XGA modes. In the first place there is no AI support for direct color mode operations. Another limitation is that in the original XGA, the graphics coprocessor is not operational in the direct color mode. Note that this restriction does not apply to the XGA-2 systems in which coprocessor support was extended to the direct color modes. The direct color mode has a resolution of 640-by-480 pixels. Each screen pixel can be in any one of 65,536 colors. The pixel's color code in stored in 16 bits; therefore each pixel is mapped to one word in the video buffer. Each screen row requires 1280 buffer bytes and the total storage is 614,400 bytes. Since a memory bank consists of 64K, 9.375 banks are used in this mode.

```
        ASSUME  CS:P_CODE
.386
DC_PALETTE      PROC    FAR
; Set 256 XGA Palette registers for the 65535-color mode
; Note: the values are those recommended by IBM
; Code assumes that XGA system is set in a graphics mode
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT    ; Local data segment
        MOV     DS,AX            ; to DS
        ASSUME  DS:XGA_DIRECT
;
; Select Index register at offset 0AH
        MOV    DX,XGA_REG_BASE   ; Base address of controller
        ADD    DX,0AH            ; To Index register
; Write 00H (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Clearing all bits
; makes the palette invisible during setup
        MOV    AX,0064H          ; Make invisible
        OUT    DX,AX
; Write 00H (in AH) to Palette Sequence register (66H) to enable
; three-color write mode (RGB) and to start with the
; R color code
        MOV     AX,0066H          ; Palette Sequence register
        OUT     DX,AX
; Write 00H (in AH) to palette Index register low (60H)
; and high (61H) to select first DAC register
        MOV     AX,0060H          ; Start at palette 0
        OUT     DX,AX
        MOV     AX,0061H          ; Sprite index high
        OUT     DX,AX
;**********************|
; first 128 registers  |
;**********************|
; Write 80H (in AH) to Border Color register (55H) to select
; first group of 128 registers
        MOV     AX,8055H          ; Border Color bit 7 set
        OUT     DX,AX
        CALL    LOAD_128          ; Local procedure
;**********************|
; second 128 registers |
;**********************|
; Write 00H (in AH) to Border Color register (55H) to select the
; second group of 128 registers
        MOV     AX,0055H         ; Border Color bit 7 clear
        OUT     DX,AX
        CALL    LOAD_128         ; Local procedure
; Write FFH (in AH) to Palette Mask register (64H)
```

## 9.2.1 The Direct Color Palette

Although the DAC registers are bypassed during direct color mode operation, the IBM documentation states that the DAC registers must be loaded with specific data for operating in the direct color mode. Table 9.1 shows the values recommended by IBM.

Table 9.1 *XGA Direct Color Mode Palette Values*

| LOCATION | BORDER COLOR BIT 7 | RED | BLUE | GREEN |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 8 |
| 2 | 1 | 0 | 0 | 16 |
| 3 | 1 | 0 | 0 | 24 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 31 | 1 | 0 | 0 | 256 |
| 32 | 1 | 0 | 0 | 0 |
| 33 | 1 | 0 | 0 | 8 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 126 | 1 | 0 | 0 | 240 |
| 127 | 1 | 0 | 0 | 248 |
| 128 | 0 | 0 | 0 | 0 |
| 129 | 0 | 0 | 0 | 8 |
| 130 | 0 | 0 | 0 | 16 |
| 131 | 0 | 0 | 0 | 24 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 159 | 0 | 0 | 0 | 256 |
| 160 | 0 | 0 | 0 | 0 |
| 161 | 0 | 0 | 0 | 8 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 254 | 0 | 0 | 0 | 240 |
| 255 | 0 | 0 | 0 | 248 |

Bit 7 of the Border Color register (at offset 55H) is used to select between the first and second group of values to be entered in the direct color palette. Notice also that the red and blue components are always zero, while the green component is incremented by eight for each successive register. The following procedure allows setting the Palette registers for the direct color mode:

```
;****************************************************************
;          data variables for XGA direct color palette
;****************************************************************
XGA_DIRECT       SEGMENT PUBLIC
;
XGA_REG_BASE     DW      0       ; Register base for XGA system
;
XGA_DIRECT       ENDS
;****************************************************************
;        processing operations for XGA direct color palette
;****************************************************************
P_CODE   SEGMENT PUBLIC
```

```
        MOV     AX,1024           ; 1024 dots per line
        MUL     DX                ; DX holds line count of address
        ADD     AX,CX             ; Plus this many dots on the line
        ADC     DX,0              ; Answer in DX:AX
                                  ; DL = bank, AX = offset
        MOV     BX,AX             ; Save offset in BX
        MOV     AX,DX             ; Move bank number to AL
;
;**********************|
;    change banks      |
;**********************|
        MOV     DX,XGA_REG_BASE ; XGA Base register address
        ADD     DX,08H            ; Aperture Index register
        OUT     DX,AL             ; Bank number is in AL
;
;**********************|
;   read the pixel     |
;**********************|
        MOV     AL,ES:[BX]        ; Read pixel into AL
        POP     DS                ; Restore caller's DS
        RET
XGA_READ_2      ENDP
P_CODE          ENDS
```

## 9.2 Programming the XGA Direct Color Mode

XGA video mode 5 is called the *direct color mode*. It consists of 640-by-480 pixels in 65,536 colors. Note that this mode is available only in XGA systems equipped with the maximum VRAM of 1Mb. The XGA direct color mode is the one with the most extensive color range. The pixel color is determined by a 16-bit value, which encodes the 65,536 colors that can be represented. The actual pixel color is generated independently of the setting of the DAC registers. For this reason the direct color mode has also been referred to as the *palette bypass mode*. The color encoding of the 16-bit value for the direct color mode is shown in Figure 9.2.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **RED (5 bits)** | | | | | **GREEN (6 bits)** | | | | | | **BLUE (5 bits)** | | | | |

Figure 9.2 *XGA Direct Color Mode Palette*

Note that the color bitmap in Figure 9.2 contains five bits for the blue and red elements and six bits for the green element. This 5-6-5 configuration allows 64 shades of green and 32 shades each of blue and red colors. The argument in favor of having more shades of green than of red and blue is that the human eye is more sensitive to the green portion of the spectrum.

## 9.1.4 Reading a Pixel

A write routine that accesses the video memory space through the CPU can be easily converted to read the value of screen pixels. The conversion consists mainly of changing the write instruction for a read instruction and in making other minor register adjustments. The following procedure can be used to read the value of a screen pixel into a CPU register:

```
;****************************************************************
;            data variables for XGA read pixel
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_REG_BASE    DW      0       ; Register base for XGA system
;
XGA_DIRECT      ENDS
;
;****************************************************************
;         processing operations for XGA read pixel
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
.386
;
XGA_READ_2       PROC     FAR
; Read a screen pixel accessing XGA memory directly
;
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
; On exit:
;       AL = pixel color
; Note: code assumes that XGA is in a 1024-by-768 pixel mode
;       in 256 colors and that A0000H is the start address for
;       the video buffer using the 64K aperture
;
;*********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT   ; Local data segment
        MOV     DS,AX           ; to DS
        ASSUME  DS:XGA_DIRECT
; Set ES to video buffer base address
        MOV     AX,0A000H       ; Base for all graphics modes
        MOV     ES,AX           ; To ES segment
; Get address in XGA system
        CLC                     ; Clear carry flag
```

```
;****************************************************************
;            data variables for XGA clear screen
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_REG_BASE    DW      0       ; Register base for XGA system
;
XGA_DIRECT      ENDS
;****************************************************************
;       processing operations for XGA clear screen
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
.386
;
XGA_CLS_2       PROC    FAR
; Clear video memory while in mode number 2 using block move
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT           ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:XGA_DIRECT
;
        MOV     AX,0A000H       ; Video memory base address
        MOV     ES,AX           ; To the ES register
        MOV     BL,0            ; BL is bank counter
; Select bank
NEXT_BANK:
        MOV     DX,XGA_REG_BASE ; Select Page
        ADD     DX,08H          ; To Aperture Index register
        MOV     AL,BL           ; Bank number
        OUT     DX,AL           ; Select bank in AL
; Write 65536 bytes of 00H in current bank
        MOV     CX,0FFFFH       ; CX is byte counter
        MOV     AX,0            ; Attribute to place in VRAM
        CLD                     ; Forward direction
        MOV     DI,0            ; Start of block
        REP     STOSB           ; Store 65536 bytes
; Bump bank
        INC     BL
        CMP     BL,12           ; 12 is last bank
        JNE     NEXT_BANK
        POP     DS              ; Restore caller's DS
        RET
XGA_CLS_2       ENDP
P_CODE          ENDS
```

```
; Write a screen pixel accessing XGA memory directly
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
;       BL = pixel color in 8-bit format
; Note: code assumes that XGA is in a 1024-by-768 pixel mode
;       in 256 colors (mode number 2)
;*********************|
;   save caller's DS  |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT           ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:XGA_DIRECT
; Set ES to video buffer base address
        MOV     AX,0A000H       ; Base for all graphics modes
        MOV     ES,AX           ; To ES segment
        MOV     AL,BL           ; Color to AL
; Get address in XGA system
        CLC                     ; Clear carry flag
        PUSH    AX              ; Save color value
        MOV     AX,1024         ; This many dots / line
        MUL     DX              ; DX holds line count of address
        ADD     AX,CX           ; Plus this many x dots
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Save this in BX
        MOV     AX,DX           ; Move bank number to AL
;*********************|
;    change banks     |
;*********************|
        MOV    DX,XGA_REG_BASE  ; XGA Base register address
        ADD    DX,08H           ; Aperture Index register
        OUT    DX,AL            ; Bank number is in AL
        POP     AX              ; Restore color value
;*********************|
;   set the pixel     |
;*********************|
        MOV     ES:[BX],AL      ; Write the dot
        POP     DS              ; Restore caller's DS
        RET
XGA_PIXEL_2     ENDP
P_CODE          ENDS
```

### 9.1.3 Clearing the XGA Screen

The following procedure uses the pixel setting routine described in Section 9.1.2 to clear the entire XGA video screen while in mode 2:

In the 64K aperture the start address for the video memory in each bank is selected by means of the Aperture Control register. The valid values are A0000H and B0000H. The first one coincides with the base address used in VGA graphics modes. If the start address of A0000H is selected, then each bank extends from A0000H to AFFFFH. Which bank is currently selected depends on the setting of the Aperture Index register, located at base address plus 8 of the XGA Display Controller group. If the base address of the Display Controller group is stored in the variable XGA_REG_BASE and the bank number is stored in the AL register, then enabling a specific bank can be coded as follows:

```
; AL holds desired memory bank
        MOV    DX,XGA_REG_BASE    ; XGA Base register address
        ADD    DX,08H             ; Aperture Index register
        OUT    DX,AL              ; Bank number is in AL
```

The total number of banks available depends on the display mode selected. We saw that 12 banks of 64K each are needed to encode all the pixels in the 1024-by-768 modes. However, in the 640-by-480 pixel mode each full screen consists of 307,200 pixels, which require only five memory banks of 64K.

### 9.1.2 Setting a Pixel

In order to set a screen pixel, the display logic must take into account whether the base address of the video buffer for the 64K aperture is located at A000H or at B000H. In addition, the code must perform the necessary bank selection operation. Processing performance in this case can be improved by storing the value of the currently selected bank in a memory variable so that bank switching can be bypassed if the pixel is located in the currently selected bank. The following code writes a data byte to a video memory address. This procedure does not take into account the currently selected bank. The code assumes XGA mode 2 in 1024-by-768 pixels in 256 colors.

```
;****************************************************************
;            data variables for XGA pixel setting
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
;
XGA_REG_BASE    DW      0       ; Register base for XGA system
;
XGA_DIRECT      ENDS
;****************************************************************
;       processing operations for XGA pixel setting
;****************************************************************
P_CODE    SEGMENT PUBLIC
          ASSUME  CS:P_CODE
;
.386
XGA_PIXEL_2    PROC   FAR
```

## 9.1 Accessing XGA Video Memory

The CPU can access XGA memory to perform write and read operations almost in the same manner as in VGA systems. The write operation sets one or more screen pixels to the value stored in a processor register. The read operation transfers a pixel's value into a processor register. We saw in Chapter 8 that the XGA system can configure video memory by means of three possible apertures. The 4Mb aperture is the one used by the graphics coprocessor, which is discussed later in this chapter. The 1Mb memory aperture is typically used in multitasking systems. The 64K aperture is the one typically used by drivers and applications executing in the MS-DOS environment.

### 9.1.1 XGA Memory Banks

DOS applications usually access XGA video memory by means of multiple memory banks of 64K each. But before the 64K aperture is available, the code must make sure that the Aperture Control register (at base address plus 1) has been initialized to the value 01H. The banks' structure at this aperture depends on the display mode. At the 1024-by-768 modes the 64K aperture can be visualized as 12 memory blocks of 64K each. This visualization is shown in Figure 9.1.

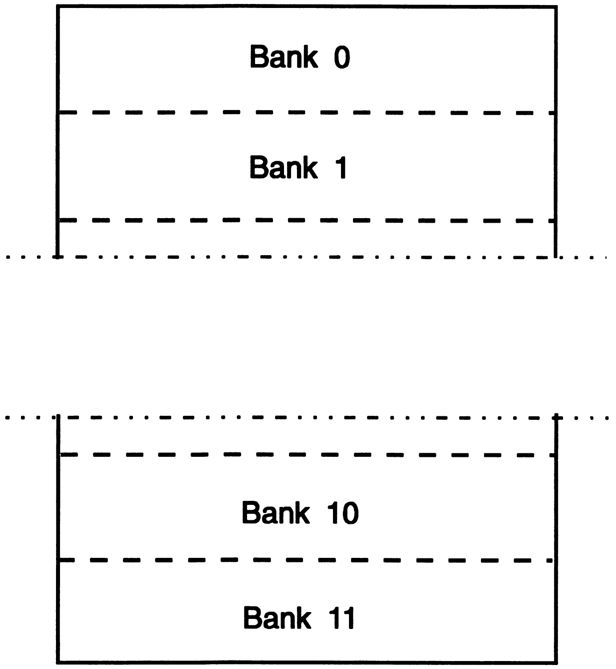


Figure 9.1 *Visualization of XGA Memory Banks*

# Chapter 9

# XGA Drivers and Primitives

## 9.0 XGA Hardware Programming

XGA systems can be programmed at the hardware and software levels. Because of its limitations and performance penalties XGA software programming by means of the Adapter Interface, by the VESA XGA BIOS, and by the XGA-2 DMQS services is not discussed in the book. This chapter is devoted to programming XGA graphics by accessing the video memory space and the coprocessor hardware.

An application can access XGA video memory through the CPU or by means of the XGA graphics coprocessor. The initial discussion relates to accessing the XGA video memory space by means of the 80386, 486, or Pentium Central Processing Unit. The animation programmer should be cautious about using CPU access methods in the XGA since, in this case, the XGA coprocessor is bypassed, resulting in a substantial performance penalty. However, there are practical circumstances that can make direct access to XGA video memory an attractive alternative; for example:

1. Direct access programming is often easier to code than XGA graphics coprocessor operations.
2. Direct access methods are usually available to the code before the coprocessor is initialized.
3. Some VGA software is easier to port to XGA system programs by means of direct access techniques than by coprocessor or AI programming.

VESA XGA functions are currently limited to providing information about the capabilities of an XGA system as well as the location of the XGA hardware and to allow setting an XGA extended graphics mode. The functions are implemented as subservice 4EH of interrupt 10H. The VESA XGA BIOS extension provides the following subservices:

1. Subservice number 0 is used to determine if the VESA XGA BIOS is present. In addition, this service provides information about the XGA environment, including the number of XGA systems detected, the version of the VESA BIOS, if bus mastering is available, and if the bus architecture is ISA, EISA, or micro channel.
2. Subservice number 1 returns XGA system information, including the address of the available apertures, the base address of the XGA CRT Controller registers, the base address of video memory, the amount of video memory installed, and the physical address of the XGA coprocessor.
3. Subservice number 2 returns XGA mode information, including the bytes per scan line, the horizontal and vertical resolution, the bits-per-pixel depth, and the number of red, green, and blue palette bits and their position in the data field.
4. Subservice number 3 is used to initialize the XGA and to set a VGA or advanced graphics mode.
5. Subservice number 4 returns the current video mode.
6. Subservice number 5 is used to enable or disable the transmission of data through the XGA feature connector and sets the direction for data transmission.
7. Subservice number 6 is used to obtain the current state of the XGA feature connector.

The VESA BIOS service can be used to detect XGA system configuration and initialize the device in a device-independent form; however, it does not provide graphics services to the caller nor does its use guarantee that the XGA hardware is compatible with IBM XGA or with any other system.

Within the limits previously mentioned, the programmer can use the DMQS or VESA BIOS services to simplify XGA initialization operations. The use of these services is described in detail in our book *High Resolution Video Graphics*, also published by McGraw-Hill (see Bibliography).

```
; In two-monitor systems the XGA image is not erased when
; switching to a VGA mode. It is left to the application to
; erase the XGA screen, if desired
TWO_MON_SYS:
        LEA     SI,VGA_L1       ; Point to start of values table
VGA2_DATA:
        MOV     DX,XGA_REG_BASE ; XGA register base
        MOV     AH,0            ; High byte of offset is 0
        MOV     AL,[SI]         ; Low byte of offset
; Register value 0FFH marks the end of the table
        CMP     AL,0FFH         ; End of the table?
        JE      VGA2_DONE       ; End of register setup
        ADD     DX,AX           ; Add register offset to base
        CMP     AL,0AH          ; Test for an Index register
        JE      INDEXED_2       ; Go if Index register
; At this point register is not at offset 0AH, and therefore data
; is output directly
        MOV     AL,[SI+2]       ; Get data value from table
        OUT     DX,AL           ; and send to port
        JMP     SHORT NEXT_REG2 ; Continue
INDEXED_2:
        MOV     AL,[SI+1]       ; Get Index register number
        MOV     AH,[SI+2]       ; Get data byte from table
        OUT     DX,AX           ; Output data to Index register
NEXT_REG2:
        ADD     SI,3            ; Index to next register in table
        JMP     VGA2_DATA
VGA2_DONE:
        POP     DS
        RET
XGA_OFF         ENDP
CODE            ENDS
```

## 8.5 Other Methods of XGA Initialization

In this chapter we have developed a procedure (named INIT_XGA) for XGA initialization. Processing is based on locating and identifying the XGA hardware by accessing the device at the register level. Later we developed another procedure (named XGA_MODE) to set the XGA graphics mode also by accessing the hardware at the register level. The program developer working on XGA-2 systems can perform these initialization and mode setting operations by using the Display Mode Query and Set (DMQS) function. This function is available as service number 31 (1FH) of BIOS interrupt 10H.

In addition, the Video Electronics Standards Association approved in 1992 an XGA extension to the VESA standard designated as VXE 1.0. The purpose of the VESA XGA extension is to define XGA registers, bits, and BIOS services in a manner that allows the coding of drivers and applications so that they are compatible with XGA systems in ISA, EISA, and micro channel machines. The

```
; Point to start of values table
; The value at offset 0 of VGA_L2 is the register number
; The value at offset 1 is the Index register number if the
; register is 0AH. The value at offset 2 is the data byte to be
; sent to the register
VGA_DATA:
        MOV     DX,XGA_REG_BASE ; XGA register base
        MOV     AH,0            ; High byte of offset is 0
        MOV     AL,[SI]         ; Low byte of offset
; Register value 0FFH marks the end of the table
        CMP     AL,0FFH         ; End of the table?
        JE      VGA1_DONE       ; End of register setup
        ADD     DX,AX           ; Add register offset to base
        CMP     AL,0AH          ; Test for an Index register
        JE      INDEXED_1       ; Go if Index register
; At this point register is not at offset 0AH, and therefore data
; is output directly
        MOV     AL,[SI+2]       ; Get data value from table
        OUT     DX,AL           ; and send to port
        JMP     SHORT NEXT_REG1 ; Continue
INDEXED_1:
        MOV     AL,[SI+1]       ; Get Index register number
        MOV     AH,[SI+2]       ; Get data byte from table
        OUT     DX,AX           ; Output data to index register
NEXT_REG1:
        ADD     SI,3            ; Index to next register in table
        JMP     VGA_DATA
;*********************|
;   enable VGA mode   |
;   (single monitor)  |
;*********************|
VGA1_DONE:
; Enable VGA graphics
        MOV     DX,03C3H        ; VGA Video Enable register
        MOV     AL,1            ; Value to enable video
        OUT     DX,AL           ; Output to port
; Select 400 scan lines using BIOS service
        MOV     AH,18           ; Service request number
        MOV     AL,2            ; Code for 400 lines
        MOV     BL,48           ; Function code
        INT     10H
; Set VGA mode currently enabled when XGA was switched on
        MOV     AH,0            ; BIOS service request
        MOV     AL,VGA_MODE     ; VGA mode from variable
        INT     10H
        POP     DS              ; Restore caller's DS
        RET
;*********************|
;    init registers   |
;   (double-monitor)  |
;*********************|
; if the system contains two monitors. The table VGA_L2 is
; also output if the system contains a single monitor
;*********************|
;   save caller's DS  |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT           ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:XGA_DIRECT
;*********************|
;   test for second   |
;       monitor       |
;*********************|
        MOV     AL,EQUIPMENT    ; Load equipment byte
        TEST    AL,00010000B    ; Bit 4 is second monitor
        JZ      ONE_MON_SYS     ; Go if bit set
        JMP     TWO_MON_SYS     ; Exit if one monitor system
;*********************|
;    single monitor   |
;        system       |
;*********************|
ONE_MON_SYS:
; Clear 256K of XGA video memory to avoid screen flash during
; mode set
        PUSH    ES              ; Save caller's ES
        MOV     AX,0A000H       ; Video memory base address
        MOV     ES,AX           ; To the ES register
        MOV     BL,0            ; BL is bank counter
; Select bank
NEXT_BANK1:
        MOV     DX,XGA_REG_BASE ; Select Page
        ADD     DX,08H          ; To Aperture Index register
        MOV     AL,BL           ; Bank number
        OUT     DX,AL           ; Select bank in AL
; Write 65536 bytes of 00H in current bank
        MOV     CX,0FFFFH       ; CX is byte counter
        MOV     AX,0            ; Attribute to place in VRAM
        CLD                     ; Forward direction
        MOV     DI,0            ; Start of block
        REP     STOSB           ; Store 65536 bytes
; Bump bank
        INC     BL
        CMP     BL,4            ; Four banks of 64K each
        JNE     NEXT_BANK1
        POP     ES              ; Restore caller's ES
;*********************|
;    init registers   |
;   (single monitor)  |
;*********************|
        LEA     SI,VGA_L2
```

The actual mode switching operation is different for a two-monitor system, in which one monitor provides the XGA graphics output and the other one the VGA signal, or for a single-monitor system, in which one monitor provides both XGA and VGA output. In the case of a two-monitor system there is also the option of clearing the XGA screen when switching to a VGA mode or leaving the XGA image on its dedicated monitor. The following code sample shows the data structures and processing required for switching from XGA to VGA modes. The code determines if it is executing in a one-monitor or a two-monitor system. Notice that the interrupt intercept and chaining operations described earlier in this section are not implemented in the code.

```
;****************************************************************
;           data variables for XGA to VGA switching
;****************************************************************
XGA_DIRECT      SEGMENT PUBLIC
; Register and data for XGA to VGA switching
; Register in VGA_L1 is always output. VGA_L2 is output if the
; XGA monitor is providing the VGA function
VGA_L1  DB    001H,000H,000H         ; Aperture Control register
        DB    004H,000H,000H         ; Interrupt disable
        DB    005H,000H,0FFH         ; Clear interrupts
        DB    0FFH,0FFH,0FFH         ; END OF LIST
;
VGA_L2  DB    001H,000H,000H         ; Aperture Control register
        DB    004H,000H,000H         ; Interrupt disable
        DB    005H,000H,0FFH         ; Clear interrupts
        DB    00AH,064H,0FFH         ; Palette Mask register
        DB    00AH,050H,015H         ; Enable VFB
        DB    00AH,050H,014H         ; Enable VFB, reset
        DB    00AH,051H,000H         ; Normal scale factors
        DB    00AH,054H,004H         ; Select VGA occilator
        DB    00AH,070H,000H         ; External VGA clock
        DB    00AH,02AH,020H         ; No Vert Sync interrupts
        DB    000H,000H,001H         ; Switch to VGA mode
        DB    0FFH,0FFH,0FFH         ; END OF LIST
;
; Variables for operational data
VGA_MODE        DB      0       ; Storage for VGA mode
;
XGA_DIRECT      ENDS
;****************************************************************
;       processing operations for XGA to VGA switching
;****************************************************************
CODE  SEGMENT PUBLIC
      ASSUME  CS:CODE
.386
XGA_OFF         PROC    FAR
; Turn off XGA and enable VGA decoding
; The table at VGA_L1 contains the values to be sent to the
; XGA registers in order to reset operation in a VGA mode,
        MOV     AX,065H         ; Select Data register
        OUT     DX,AL
        INC     DX              ; Point to first register
; DS:SI - table of palette colors
        POP     DS              ; Restore caller's DS
; Loop to send 4 blocks of 256 byte each to port 065H
NEW_PALETTE:
        MOV     AL,[SI]         ; Get byte from table
        OUT     DX,AL           ; Send to port
        INC     SI              ; Bump table pointer
        LOOP    NEW_PALETTE
;
        DEC     DX              ; Back to Select register
; Write FFH (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Setting all bits
; makes the palette visible again
        MOV     AX,0FF64H       ; All bits set
        OUT     DX,AX           ; To make visible
        RET
XGA_PALETTE     ENDP
CODE            ENDS
```

## 8.4.5 Switching from XGA to VGA Modes

XGA software often needs to switch back to a VGA display mode. In a two-monitor system the XGA image need not be erased from the corresponding monitor. If the XGA hardware is also providing the VGA function then software running in the MS-DOS environment must execute the following special provisions:

1. Intercept all calls to the BIOS video services at interrupt 10H. The intercept routine must filter those calls that can be ignored from those that must be honored. Service number 0 (set mode) must always be honored. Service number 15 (return video state) must return the current mode as 7FH.
2. Intercept all calls to the MS-DOS Critical Error Handler at interrupt 24H. In this case the application can take one of two action: first, the intercept routine can restore the VGA signal and chain to the original error handler; and second, the application can take over the error handler function entirely and display the message.
3. Intercept calls to MS-DOS Ctrl+Break hot key. Options in this case are the same as for the critical error handler.
4. Intercept all calls to the MS-DOS Program Terminate functions at interrupt 21H (service number 76), at interrupt 20H, and at interrupt 21H (service numbers 0 and 49). In these cases the application must set the XGA in a VGA text mode and chain to the original interrupt handler.

Note that some interceptions are necessary because BIOS and MS-DOS use VGA to output the error messages. Since no VGA display is active when XGA is in a proprietary mode, the error handler message would be lost if no interception were implemented.

To avoid screen garbage, it is convenient to turn off the display function while loading a new palette. The following procedure assumes that the caller has available a set of 256 palette entries in 4-byte format, such as the ones listed.

```
;****************************************************************
;                        XGA palette loading
;****************************************************************
CODE    SEGMENT PUBLIC
        ASSUME  CS:CODE
.386
;
XGA_PALETTE     PROC    FAR
; Set 256 XGA Pallete registers
; On entry:
;        DS:SI -> 1024-byte color table in RGBx format
;
; Code assumes that XGA system is in a graphics mode
;
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        MOV     AX,XGA_DIRECT  ; Local data segment
        MOV     DS,AX          ; to DS
        ASSUME  DS:XGA_DIRECT
;
; Select Index register at offset 0AH
        MOV    DX,XGA_REG_BASE ; Base address of Controller
        ADD    DX,0AH
; Write 00H (in AH) to Palette Mask register (64H)
; This value is ANDed with display memory. Clearing all bits
; makes the palette invisible during setup
        MOV    AX,0064H        ; make invisible
        OUT    DX,AX
; Write 00H (in AH) to Border Color register (55H)
        MOV    AX,0055H        ; Border Color register
        OUT    DX,AX
; Write 00000100B (in AH) to Palette Sequence register (66H) to
; select four-color write mode (RGBx) and to start with the
; R color code
        MOV     AX,0466H        ; Palette Sequence register
        OUT     DX,AX
; Write 00H (in AH) to Palette Index Register low (60H)
; and high (61H) to select first DAC register
        MOV     AX,0060H        ; start at palette 0
        OUT     DX,AX
        MOV     AX,0061H        ; Sprite index high
        OUT     DX,AX
        MOV     CX,1024         ; Counter for 256 * 4
```

```
                    DB      168,000,128,000,168,084,128,000 ; 32-33
                    DB      168,168,128,000,168,252,128,000 ;
                    DB      168,000,144,000,168,084,144,000 ;
                    DB      168,168,144,000,168,252,144,000 ;
                    DB      168,000,160,000,168,084,160,000 ;
                    DB      168,168,160,000,168,252,160,000 ;
                    DB      168,000,176,000,168,084,176,000 ;
                    DB      168,168,176,000,168,252,176,000 ;
                    DB      168,000,192,000,168,084,192,000 ; 47-48
                    DB      168,168,192,000,168,252,192,000 ;
                    DB      168,000,208,000,168,084,208,000 ;
                    DB      168,168,208,000,168,252,208,000 ;
                    DB      168,000,224,000,168,084,224,000 ;
                    DB      168,168,224,000,168,252,224,000 ;
                    DB      168,000,240,000,168,084,240,000 ;
                    DB      168,168,240,000,168,252,240,000 ; 62-63
;
                    DB      252,000,000,000,252,084,000,000 ; 0-1
                    DB      252,168,000,000,252,252,000,000 ;
                    DB      252,000,015,000,252,084,015,000 ;
                    DB      252,168,015,000,252,252,015,000 ;
                    DB      252,000,030,000,252,084,030,000 ;
                    DB      252,168,030,000,252,252,030,000 ;
                    DB      252,000,048,000,252,084,048,000 ;
                    DB      252,168,048,000,252,252,048,000 ;
                    DB      252,000,064,000,252,084,064,000 ; 16-17
                    DB      252,168,064,000,252,252,064,000 ;
                    DB      252,000,080,000,252,084,080,000 ;
                    DB      252,168,080,000,252,252,080,000 ;
                    DB      252,000,096,000,252,084,096,000 ;
                    DB      252,168,096,000,252,252,096,000 ;
                    DB      252,000,112,000,252,084,112,000 ;
                    DB      252,168,112,000,252,252,112,000 ;
                    DB      252,000,128,000,252,084,128,000 ; 32-33
                    DB      252,168,128,000,252,252,128,000 ;
                    DB      252,000,144,000,252,084,144,000 ;
                    DB      252,168,144,000,252,252,144,000 ;
                    DB      252,000,160,000,252,084,160,000 ;
                    DB      252,168,160,000,252,252,160,000 ;
                    DB      252,000,176,000,252,084,176,000 ;
                    DB      252,168,176,000,252,252,176,000 ;
                    DB      252,000,192,000,252,084,192,000 ; 47-48
                    DB      252,168,192,000,252,252,192,000 ;
                    DB      252,000,208,000,252,084,208,000 ;
                    DB      252,168,208,000,252,252,208,000 ;
                    DB      252,000,224,000,252,084,224,000 ;
                    DB      252,168,224,000,252,252,224,000 ;
                    DB      252,000,240,000,252,084,240,000 ;
                    DB      252,168,240,000,252,252,240,000 ; 62-63
;
DATA                ENDS
```

```
DB      000,168,240,000,000,252,240,000 ; 62-63

DB      084,000,000,000,084,084,000,000 ; 0-1
DB      084,168,000,000,084,252,000,000 ;
DB      084,000,015,000,084,084,015,000 ;
DB      084,168,015,000,084,252,015,000 ;
DB      084,000,030,000,084,084,030,000 ;
DB      084,168,030,000,084,252,030,000 ;
DB      084,000,048,000,084,084,048,000 ;
DB      084,168,048,000,084,252,048,000 ;
DB      084,000,064,000,084,084,064,000 ; 16-17
DB      084,168,064,000,084,252,064,000 ;
DB      084,000,080,000,084,084,080,000 ;
DB      084,168,080,000,084,252,080,000 ;
DB      084,000,096,000,084,084,096,000 ;
DB      084,168,096,000,084,252,096,000 ;
DB      084,000,112,000,084,084,112,000 ;
DB      084,168,112,000,084,252,112,000 ;
DB      084,000,128,000,084,084,128,000 ; 32-33
DB      084,168,128,000,084,252,128,000 ;
DB      084,000,144,000,084,084,144,000 ;
DB      084,168,144,000,084,252,144,000 ;
DB      084,000,160,000,084,084,160,000 ;
DB      084,168,160,000,084,252,160,000 ;
DB      084,000,176,000,084,084,176,000 ;
DB      084,168,176,000,084,252,176,000 ;
DB      084,000,192,000,084,084,192,000 ; 47-48
DB      084,168,192,000,084,252,192,000 ;
DB      084,000,208,000,084,084,208,000 ;
DB      084,168,208,000,084,252,208,000 ;
DB      084,000,224,000,084,084,224,000 ;
DB      084,168,224,000,084,252,224,000 ;
DB      084,000,240,000,084,084,240,000 ;
DB      084,168,240,000,084,252,240,000 ; 62-63

DB      168,000,000,000,168,084,000,000 ; 0-1
DB      168,168,000,000,168,252,000,000 ;
DB      168,000,015,000,168,084,015,000 ;
DB      168,168,015,000,168,252,015,000 ;
DB      168,000,030,000,168,084,030,000 ;
DB      168,168,030,000,168,252,030,000 ;
DB      168,000,048,000,168,084,048,000 ;
DB      168,168,048,000,168,252,048,000 ;
DB      168,000,064,000,168,084,064,000 ; 16-17
DB      168,168,064,000,168,252,064,000 ;
DB      168,000,080,000,168,084,080,000 ;
DB      168,168,080,000,168,252,080,000 ;
DB      168,000,096,000,168,084,096,000 ;
DB      168,168,096,000,168,252,096,000 ;
DB      168,000,112,000,168,084,112,000 ;
DB      168,168,112,000,168,252,112,000 ;
```

```
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
;*****************************************************************
;                    IBM-recommended palette
;*****************************************************************
; IBM-recommended palette in RR GGGG BB format
;           7 6 5 4 3 2 1 0  <== Bits
;           R R G G G G B B  <== Color codes
;
;                     | R   B   G       R   B   G     |
; to 00 0000 11
IBM1_PAL        DB      000,000,000,000,000,084,000,000 ; 0-1
                DB      000,168,000,000,000,252,000,000 ;
                DB      000,000,015,000,000,084,015,000 ;
                DB      000,168,015,000,000,252,015,000 ;
                DB      000,000,030,000,000,084,030,000 ;
                DB      000,168,030,000,000,252,030,000 ;
                DB      000,000,048,000,000,084,048,000 ;
                DB      000,168,048,000,000,252,048,000 ;
                DB      000,000,064,000,000,084,064,000 ; 16-17
                DB      000,168,064,000,000,252,064,000 ;
                DB      000,000,080,000,000,084,080,000 ;
                DB      000,168,080,000,000,252,080,000 ;
                DB      000,000,096,000,000,084,096,000 ;
                DB      000,168,096,000,000,252,096,000 ;
                DB      000,000,112,000,000,084,112,000 ;
                DB      000,168,112,000,000,252,112,000 ;
                DB      000,000,128,000,000,084,128,000 ; 32-33
                DB      000,168,128,000,000,252,128,000 ;
                DB      000,000,144,000,000,084,144,000 ;
                DB      000,168,144,000,000,252,144,000 ;
                DB      000,000,160,000,000,084,160,000 ;
                DB      000,168,160,000,000,252,160,000 ;
                DB      000,000,176,000,000,084,176,000 ;
                DB      000,168,176,000,000,252,176,000 ;
                DB      000,000,192,000,000,084,192,000 ; 47-48
                DB      000,168,192,000,000,252,192,000 ;
                DB      000,000,208,000,000,084,208,000 ;
                DB      000,168,208,000,000,252,208,000 ;
                DB      000,000,224,000,000,084,224,000 ;
                DB      000,168,224,000,000,252,224,000 ;
                DB      000,000,240,000,000,084,240,000 ;
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
                DB      152,152,152,000,156,156,156,000 ; 39
                DB      160,160,160,000,164,164,164,000 ; 41
                DB      168,168,168,000,172,172,172,000 ; 43
                DB      176,176,176,000,180,180,180,000 ; 45
                DB      184,184,184,000,188,188,188,000 ; 47
                DB      192,192,192,000,196,196,196,000 ; 49
                DB      200,200,200,000,204,204,204,000 ; 51
                DB      208,208,208,000,212,212,212,000 ; 53
                DB      216,216,216,000,220,220,220,000 ; 55
                DB      224,224,224,000,228,228,228,000 ; 57
                DB      232,232,232,000,236,236,236,000 ; 59
                DB      240,240,240,000,244,244,244,000 ; 61
                DB      248,248,248,000,252,252,252,000 ; 63
;
                DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      144,144,216,000,144,180,216,000 ; 9
                DB      144,216,216,000,144,252,216,000 ; 11
                DB      144,144,252,000,144,180,252,000 ; 13
                DB      144,216,252,000,144,252,252,000 ; 15
                DB      180,144,144,000,180,180,144,000 ; 17
                DB      180,216,144,000,180,252,144,000 ; 19
                DB      180,144,180,000,180,180,180,000 ; 21
                DB      180,216,180,000,180,252,180,000 ; 23
                DB      180,144,216,000,180,180,216,000 ; 25
                DB      180,216,216,000,180,252,216,000 ; 27
                DB      180,144,252,000,180,180,252,000 ; 29
                DB      180,216,252,000,180,252,252,000 ; 31
                DB      216,144,144,000,216,180,144,000 ; 33
                DB      216,215,144,000,216,252,144,000 ; 35
                DB      216,144,180,000,216,180,180,000 ; 37
                DB      216,216,180,000,216,252,180,000 ; 39
                DB      216,144,216,000,216,180,216,000 ; 41
                DB      216,216,216,000,216,252,216,000 ; 43
                DB      216,144,252,000,216,180,252,000 ; 45
                DB      216,216,252,000,216,252,252,000 ; 47
                DB      252,144,144,000,252,180,144,000 ; 49
                DB      252,216,144,000,252,252,144,000 ; 51
                DB      252,144,180,000,252,180,180,000 ; 53
                DB      252,216,180,000,252,252,180,000 ; 55
                DB      252,144,216,000,252,180,216,000 ; 57
                DB      252,216,216,000,252,252,216,000 ; 59
                DB      252,144,252,000,252,180,252,000 ; 61
                DB      252,216,252,000,252,252,252,000 ; 63
;
;****************************************************************
;                       Grayscale palette
;****************************************************************
GRAY_PAL        DB      000,000,000,000,004,004,004,000 ; 1
                DB      008,008,008,000,012,012,012,000 ; 3
                DB      016,016,016,000,020,020,020,000 ; 5
                DB      024,024,024,000,028,028,028,000 ; 7
                DB      032,032,032,000,036,036,036,000 ; 9
                DB      040,040,040,000,044,044,044,000 ; 11
                DB      048,048,048,000,052,052,052,000 ; 13
                DB      056,056,056,000,060,060,060,000 ; 15
                DB      064,064,064,000,068,068,068,000 ; 17
                DB      072,072,072,000,076,076,076,000 ; 19
                DB      080,080,080,000,084,084,084,000 ; 21
                DB      088,088,088,000,092,092,092,000 ; 23
                DB      096,096,096,000,100,100,100,000 ; 25
                DB      104,104,104,000,108,108,108,000 ; 27
                DB      112,112,112,000,116,116,116,000 ; 29
                DB      120,120,120,000,124,124,124,000 ; 31
                DB      128,128,128,000,132,132,132,000 ; 33
                DB      136,136,136,000,140,140,140,000 ; 35
                DB      144,144,144,000,148,148,148,000 ; 37
DB      144,144,108,000,144,180,108,000 ; 39
DB      144,072,144,000,144,108,144,000 ; 41
DB      144,144,144,000,144,180,144,000 ; 43
DB      144,072,180,000,144,108,180,000 ; 45
DB      144,144,180,000,144,180,180,000 ; 47
DB      180,072,072,000,180,108,072,000 ; 49
DB      180,144,072,000,180,180,072,000 ; 51
DB      180,072,108,000,180,108,108,000 ; 53
DB      180,144,108,000,180,180,108,000 ; 55
DB      180,072,144,000,180,108,144,000 ; 57
DB      180,144,144,000,180,180,144,000 ; 59
DB      180,072,180,000,180,108,180,000 ; 61
DB      180,144,180,000,180,180,180,000 ; 63

DB      108,108,108,000,108,144,108,000 ; 1
DB      108,180,108,000,108,216,108,000 ; 3
DB      108,108,144,000,108,144,144,000 ; 5
DB      108,180,144,000,108,216,144,000 ; 7
DB      108,108,180,000,108,144,180,000 ; 9
DB      108,180,180,000,108,216,180,000 ; 11
DB      108,108,216,000,108,144,216,000 ; 13
DB      108,180,216,000,108,216,216,000 ; 15
DB      144,108,108,000,144,144,108,000 ; 17
DB      144,180,108,000,144,216,108,000 ; 19
DB      144,108,144,000,144,144,144,000 ; 21
DB      144,180,144,000,144,216,144,000 ; 23
DB      144,108,180,000,144,144,180,000 ; 25
DB      144,180,180,000,144,216,180,000 ; 27
DB      144,108,216,000,144,144,216,000 ; 29
DB      144,180,216,000,144,216,216,000 ; 31
DB      180,108,108,000,180,144,108,000 ; 33
DB      180,180,108,000,180,216,108,000 ; 35
DB      180,108,144,000,180,144,144,000 ; 37
DB      180,180,144,000,180,216,144,000 ; 39
DB      180,108,180,000,180,144,180,000 ; 41
DB      180,180,180,000,180,216,180,000 ; 43
DB      180,108,216,000,180,144,216,000 ; 45
DB      180,180,216,000,180,216,216,000 ; 47
DB      216,108,108,000,216,144,108,000 ; 49
DB      216,180,108,000,216,216,108,000 ; 51
DB      216,108,144,000,216,144,144,000 ; 53
DB      216,180,144,000,216,216,144,000 ; 55
DB      216,108,180,000,216,144,180,000 ; 57
DB      216,180,180,000,216,216,180,000 ; 59
DB      216,108,216,000,216,144,216,000 ; 61
DB      216,180,216,000,216,216,216,000 ; 63

DB      144,144,144,000,144,180,144,000 ; 1
DB      144,216,144,000,144,252,144,000 ; 3
DB      144,144,180,000,144,180,180,000 ; 5
DB      144,216,180,000,144,252,180,000 ; 7
```

```
                DB      036,108,036,000,036,144,036,000 ; 3
                DB      036,036,072,000,036,072,072,000 ; 5
                DB      036,108,072,000,036,144,072,000 ; 7
                DB      036,036,108,000,036,072,108,000 ; 9
                DB      036,108,108,000,036,144,108,000 ; 11
                DB      036,036,144,000,036,072,144,000 ; 13
                DB      036,108,144,000,036,144,144,000 ; 15
                DB      072,036,036,000,072,072,036,000 ; 17
                DB      072,108,036,000,072,144,036,000 ; 19
                DB      072,036,072,000,072,072,072,000 ; 21
                DB      072,108,072,000,072,144,072,000 ; 23
                DB      072,036,108,000,072,072,108,000 ; 25
                DB      072,108,108,000,072,144,108,000 ; 27
                DB      072,036,144,000,072,072,144,000 ; 29
                DB      072,108,144,000,072,144,144,000 ; 31
                DB      108,036,036,000,108,071,036,000 ; 33
                DB      108,108,036,000,108,144,036,000 ; 35
                DB      108,036,072,000,108,072,072,000 ; 37
                DB      108,108,072,000,108,144,072,000 ; 39
                DB      108,036,108,000,108,072,108,000 ; 41
                DB      108,108,108,000,108,144,108,000 ; 43
                DB      108,036,144,000,108,072,144,000 ; 45
                DB      108,108,144,000,108,144,144,000 ; 47
                DB      144,036,036,000,144,072,036,000 ; 49
                DB      144,108,036,000,144,144,036,000 ; 51
                DB      144,036,072,000,144,072,072,000 ; 53
                DB      144,108,072,000,144,144,072,000 ; 55
                DB      144,036,108,000,144,072,108,000 ; 57
                DB      144,108,108,000,144,144,108,000 ; 59
                DB      144,036,144,000,144,072,144,000 ; 61
                DB      144,108,144,000,144,144,144,000 ; 63
;
                DB      072,072,072,000,072,108,072,000 ; 1
                DB      072,144,072,000,072,180,072,000 ; 3
                DB      072,072,108,000,072,108,108,000 ; 5
                DB      072,144,108,000,072,180,108,000 ; 7
                DB      072,072,144,000,072,108,144,000 ; 9
                DB      072,144,144,000,072,180,144,000 ; 11
                DB      072,072,180,000,072,108,180,000 ; 13
                DB      072,144,180,000,072,180,180,000 ; 15
                DB      108,072,072,000,108,108,072,000 ; 17
                DB      108,144,072,000,108,180,072,000 ; 19
                DB      108,072,108,000,108,108,108,000 ; 21
                DB      108,144,108,000,108,180,108,000 ; 23
                DB      108,072,144,000,108,108,144,000 ; 25
                DB      108,144,144,000,108,180,144,000 ; 27
                DB      108,072,180,000,108,108,180,000 ; 29
                DB      108,144,180,000,108,180,180,000 ; 31
                DB      144,072,072,000,144,108,072,000 ; 33
                DB      144,144,072,000,144,180,072,000 ; 35
                DB      144,072,108,000,144,108,108,000 ; 37
```

in groups of three items representing the red, blue, and green colors. In the 4-value update mode data is written in groups of four items; the first three represent the red, blue, and green values, and the fourth item is a padding byte which is ignored by the hardware. The 3-value sequence is similar to the one used in VGA systems. The 4-value sequence is the one used by the AI. The update mode is selected by means of bit 2 of the Palette Sequence register.

Notice that in the XGA palette the six high-order bits are significant while in VGA the significant bits are the 6 low ones. Also notice that in the XGA-2 upgrade the Pallete registers have been expanded to 8 bits. Figure 8.9 shows the Pallete registers in VGA, XGA, and XGA-2 systems.

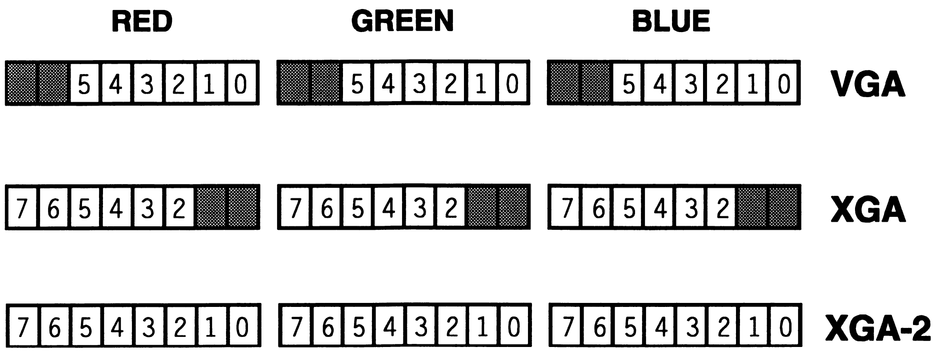


Figure 8.9 *DAC Register Bitmaps for VGA, XGA, and XGA-2*

## Loading the XGA Palette

Palette data for a full XGA palette consists of 256 red, green, and blue values (color triplets), one for each Pallete register. We have mentioned that the color values can be encoded in sets of three or four. The four-entry format is compatible with the one used in the AI.

The following code fragment shows three listings for XGA palettes. The first one, named IRGB_PAL, is a palette in double-bit IRGB format. The second palette, named GRAY_PAL, is a grayscale palette that provides monochrome settings for all DAC registers. The third palette, named IBM1_PAL, uses the DAC register settings recommended in IBM documentation.

```
DATA            SEGMENT
;
;*****************************************************************
;                         IRGB palette
;*****************************************************************
; Double-bit IRGB palette in the following format
;           7 6 5 4 3 2 1 0  <== Bits
;           I I R R G G B B  <== Color codes
;                     | R   B   G       R   B   G     |
IRGB_PAL        DB      000,000,000,000,036,072,036,000 ; 1
```

Table 8.6 *Default Setting of XGA LUT Registers*

| REGISTER NUMBER | 6-BIT COLOR (HEX VALUE) R | G | B | COLOR |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Black |
| 1 | 0 | 0 | 168 | Dark blue |
| 2 | 0 | 168 | 0 | Dark green |
| 3 | 0 | 168 | 168 | Dark cyan |
| 4 | 168 | 0 | 0 | Dark red |
| 5 | 168 | 0 | 168 | Dark magenta |
| 6 | 168 | 84 | 00 | Brown |
| 7 | 168 | 168 | 168 | Gray |
| 8 | 84 | 84 | 84 | Dark gray |
| 9 | 84 | 84 | 252 | Light blue |
| 10 | 84 | 252 | 84 | Light green |
| 11 | 84 | 252 | 252 | Light cyan |
| 12 | 252 | 84 | 84 | Light red |
| 13 | 252 | 84 | 252 | Light magenta |
| 14 | 252 | 252 | 84 | Yellow |
| 15 | 252 | 252 | 252 | Bright white |
| 16 to 31 | 0 | 0 | 168 | Dark blue |
| 32 to 47 | 0 | 168 | 0 | Dark green |
| 48 to 63 | 0 | 168 | 168 | Dark cyan |
| 64 to 79 | 168 | 0 | 0 | Dark red |
| 80 to 95 | 168 | 0 | 168 | Dark magenta |
| 96 to 111 | 168 | 84 | 0 | Brown |
| 112 to 127 | 168 | 168 | 168 | Gray |
| 128 to 143 | 84 | 84 | 84 | Dark gray |
| 144 to 849 | 84 | 84 | 252 | Light blue |
| 160 to 175 | 84 | 252 | 84 | Light green |
| 176 to 191 | 84 | 252 | 252 | Light cyan |
| 192 to 207 | 252 | 84 | 84 | Light red |
| 208 to 223 | 252 | 84 | 252 | Light magenta |
| 224 to 239 | 252 | 252 | 84 | Yellow |
| 240 to 255 | 252 | 252 | 252 | Bright white |

Notice that the default settings for the XGA registers represent only 16 color values, which correspond to registers 0 to 15 in Table 8.6. The default colors encoded in LUT registers 16 to 255 are but a repetition, in groups of 16 registers, of the encodings in the first 16 LUT registers. Consequently, software products that intend to use the full color range of the XGA must reset the Palette registers.

## Palette Structure

The XGA palette data consists of red, blue, and green values, sometimes called *color triplets*, that are stored in corresponding registers. The mechanism resembles the one used by the VGA palette in the 256-color modes. However, the XGA palette is a simpler device than the one in VGA since no Palette or Color Select registers are involved. The XGA palette consists of three sets of 256 registers in which the red, blue, and green DAC values are stored. In this manner a pixel color can be interpreted as a Palette register number; the actual color in which the pixel is displayed depends on the value stored in the corresponding Palette register.

The XGA Palette register hardware consists of 256 locations; each location is divided into three fields. The first field corresponds to the red DAC value, the second one to the blue, and the third field to the green. The XGA allows two update mode: in the 3-value update mode data is written to the Pallete registers

```
;   clear screen and   |
; restore video signal |
;**********************|
SETUP_END:
; At this point the screen should be cleared to avoid
; a disturbing screen flash when the VGA video signal is
; restored. Since the video buffer in VGA modes is 256K,
; this is the memory area that must be cleared. The operation is
; preformed by the X_CLEAR_256 procedure in the XGA3 module
        CALL    X_CLEAR_256     ; Library procedure in XGA3
; Make palette visible to restore video
        MOV     DX,XGA_REG_BASE
        ADD     DX,0AH          ; Index register
        MOV     AX,0FF64H       ; Value is all ones for ON
        OUT     DX,AX           ; Write data
; Exit
        CLC                     ; No error
        POP     DS              ; Restore caller;'s DS
        RET
BAD_MODE:
        STC                     ; Carry is error flag
        POP     DS              ; Restore caller;'s DS
        RET
XGA_MODE        ENDP
CODE            ENDS
```

## 8.4.4 The XGA Palette

We have seen that XGA video memory is organized in bit planes. Each bit plane encodes the color for a rectangular array of 1024-by-1024 pixels. Since the highest available resolution is 1024-by-768 pixels, there are 256 unused bits in each plane. When the graphics system is in a low-resolution mode, video memory consist of eight 1024-by-512 bit planes, which are divided into two separate groups of four bit planes each. These two bit-plane groups can be simultaneously addressed. In the low-resolution mode the color range is limited to 16 simultaneous colors. In the high-resolution mode video memory consists of eight bit planes of 1024-by-1024 pixels. In this mode the number of simultaneous colors is 256. Figure 8.2 shows the bit-plane mapping in XGA high-resolution modes.

### Color Look-up Table

Color selection is performed by means of a color look-up table (LUT) associated with the XGA DAC. The selection mechanism is similar to the one used in VGA mode number 19; that is, the 8-bit color code stored in XGA video memory serves as an index into the color look-up table. For example, the color value 12 in video memory selects LUT register number 12, which in the default setting stores the encoding for light red. The default settings of the LUT registers can be seen in Table 8.6.

```
;**********************|
;  graphics mode setup |
;**********************|
VALID_MODE:
; Enable VGA graphics
; Note: IBM documentation recommends using BIOS service 12H
;       (BL=32H) to enable and disable VGA video
        MOV     DX,03C3H        ; VGA Video Enable register
        MOV     AL,1            ; Value to enable video
        OUT     DX,AL           ; Output to port
; Store current VGA video mode
        MOV     AH,15           ; BIOS service request number
        INT     10H
        MOV     VGA_MODE,AL     ; Store it for VGA reset
;
; The table at XGA_VAL contains the values to be sent to the
; XGA register in order to initialize the corresponding mode
        LEA     SI,XGA_VAL      ; Point to start of values table
        MOV     BX,MODE         ; Use mode as an offset
;**********************|
;    initialize XGA    |
;       registers      |
;**********************|
; The value at offset 0 of XGA_VAL is the register number
; The value at offset 1 is the index register number if the
; register is 0AH. The remaining entries are register data for
; each mode
REG_DATA:
        MOV     DX,XGA_REG_BASE ; XGA register base
        MOV     AH,0            ; High byte of offset is 0
        MOV     AL,[SI]         ; Low byte of offset
; Register value 0FFH marks the end of the table
        CMP     AL,0FFH         ; End of the table?
        JE      SETUP_END       ; End of register setup
        ADD     DX,AX           ; Add register offset to base
        CMP     AL,0AH          ; Test for an Index register
        JE      INDEXED         ; Go if Index register
; At this point register is not at offset 0AH, therefore data
; is output directly
; BX holds mode number, which is offset into table
        MOV     AL,[SI+BX]      ; Get data value from table
        OUT     DX,AL           ; and send to port
        JMP     SHORT NEXT_REG  ; Continue
INDEXED:
        MOV     AL,[SI+1]       ; Get Index register number
        MOV     AH,[SI+BX]      ; Get data byte from table
        OUT     DX,AX           ; Output data to Index register
NEXT_REG:
        ADD     SI,6             ; Index to next register
        JMP     REG_DATA
;**********************|
        DB    00AH,050H,00FH,00FH,0C7H,0C7H ; Display mode 1
        DB    00AH,055H,000H,000H,000H,000H ; Border color
        DB    00AH,060H,000H,000H,000H,000H ; Sprite pal lo
        DB    00AH,061H,000H,000H,000H,000H ; Sprite pal hi
        DB    00AH,062H,000H,000H,000H,000H ; Sprite pre lo
        DB    00AH,063H,000H,000H,000H,000H ; Sprite pre hi
        DB    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH ; End of the list
; Note: the Palette Mask register at offset 64H must be set to
;       FFH to reenable the XGA signal
;
XGA_DIRECT      ENDS
;
;****************************************************************
;       processing operations for XGA mode setting
;****************************************************************
CODE  SEGMENT PUBLIC
      ASSUME  CS:CODE
.386
;
XGA_MODE        PROC     FAR
; Procedure to initialize an XGA graphics mode by setting the
; video system registers directly
;
; On entry:
;          AL = mode number (valid range is 2 to 5)
;
; On exit:
;         carry clear if no error
;
;**********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;**********************|
        PUSH    DS
        PUSH    AX                ; Save caller's mode
        MOV     AX,XGA_DIRECT     ; Local data segment
        MOV     DS,AX             ; to DS
        ASSUME  DS:XGA_DIRECT
        POP     AX                ; Restore caller's mode
        MOV     AH,0              ; Clear high mode byte
        MOV     MODE,AX           ; Mode to variable
        CMP     MODE,6            ; Mode number out of range?
        JB      TEST_MODE1        ; Go if less than 6
        JMP     BAD_MODE          ; illegal entry value for mode
; Mode 0 = VGA BIOS mode number 3
; Mode 1 = 132 column text mode
; These modes are not valid
TEST_MODE1:
        CMP     MODE,1            ; 80-col VGA text mode?
        JA      VALID_MODE        ; Go if range is  1
        JMP     BAD_MODE          ; Error exit for invalid mode
```

```
; 1024x768x256 ----------|    |    |    |
;                        |    |    |    |
; Index ------------|    |    |    |    |
; Register-----|    |    |    |    |    |
;             _|__ _|_  _|__ _|__ _|__ _|__
XGA_VAL DB    004H,000H,000H,000H,000H,000H ; Interrupt enable
        DB    005H,000H,0FFH,0FFH,0FFH,0FFH ; Interrupt status
        DB    000H,000H,004H,004H,004H,004H ; Operating mode
        DB    00AH,064H,000H,000H,000H,000H ; Palette mask
        DB    001H,000H,001H,001H,001H,001H ; Vid mem aper cntl
        DB    008H,000H,000H,000H,000H,000H ; Vid mem aper indx
        DB    006H,000H,000H,000H,000H,000H ; Virt mem ctl
        DB    009H,000H,003H,002H,003H,004H ; Mem access mode
        DB    00AH,050H,001H,001H,001H,001H ; Disp mode 1
        DB    00AH,050H,000H,000H,000H,000H ; Disp mode 1
        DB    00AH,010H,09DH,09DH,063H,063H ; Horiz tot lo.
        DB    00AH,011H,000H,000H,000H,000H ; Horiz tot hi.
        DB    00AH,012H,07FH,07FH,04FH,04FH ; Hor disp end lo
        DB    00AH,013H,000H,000H,000H,000H ; Hor disp end hi
        DB    00AH,014H,07FH,07FH,04FH,04FH ; Hor blank start lo
        DB    00AH,015H,000H,000H,000H,000H ; Hor blank start hi
        DB    00AH,016H,09DH,09DH,063H,063H ; Hor blank end lo
        DB    00AH,017H,000H,000H,000H,000H ; Hor blank end hi
        DB    00AH,018H,087H,087H,055H,055H ; Hor sync start lo
        DB    00AH,019H,000H,000H,000H,000H ; Hor sync start hi
        DB    00AH,01AH,09CH,09CH,061H,061H ; Hor sync end lo
        DB    00AH,01BH,000H,000H,000H,000H ; Hor sync end hi
        DB    00AH,01CH,040H,040H,000H,000H ; Hor sync pos
        DB    00AH,01EH,004H,004H,000H,000H ; Hor sync pos
        DB    00AH,020H,030H,030H,00CH,00CH ; Vert tot lo
        DB    00AH,021H,003H,003H,002H,002H ; Vert tot hi
        DB    00AH,022H,0FFH,0FFH,0DFH,0DFH ; Vert disp end lo
        DB    00AH,023H,002H,002H,001H,001H ; Vert disp end hi
        DB    00AH,024H,0FFH,0FFH,0DFH,0DFH ; Vert blank start lo
        DB    00AH,025H,002H,002H,001H,001H ; Vert blank start hi
        DB    00AH,026H,030H,030H,00CH,00CH ; Vert blank end lo
        DB    00AH,027H,003H,003H,002H,002H ; Vert blank end hi
        DB    00AH,028H,000H,000H,0EAH,0EAH ; Vert sync start lo
        DB    00AH,029H,003H,003H,001H,001H ; Vert sync start hi
        DB    00AH,02AH,008H,008H,0ECH,0ECH ; Vert sync end
        DB    00AH,02CH,0FFH,0FFH,0FFH,0FFH ; Vert line comp lo
        DB    00AH,02DH,0FFH,0FFH,0FFH,0FFH ; Vert line comp hi
        DB    00AH,036H,000H,000H,000H,000H ; Sprite cntl
        DB    00AH,040H,000H,000H,000H,000H ; Start addr lo
        DB    00AH,041H,000H,000H,000H,000H ; Start addr me
        DB    00AH,042H,000H,000H,000H,000H ; Start addr hi
        DB    00AH,043H,080H,040H,050H,0A0H ; Pixel map width lo
        DB    00AH,044H,000H,000H,000H,000H ; Pixel map width hi
        DB    00AH,054H,00DH,00DH,000H,000H ; Clock sel
        DB    00AH,051H,003H,002H,003H,004H ; Display mode 2
        DB    00AH,070H,000H,000H,000H,000H ; Ext clock sel
```

```
HIGH_RES_TESTS:
        TEST    BL,00001000B    ; Bit 3 set if 1Mb RAM
        JNZ     DC_OR_256       ; Go to DC or 256-color options
; There is 512K RAM. Best mode is mode number 3
        MOV     DX,3            ; Force mode 3
        JMP     BEST_MODE_EXIT
DC_OR_256:
        CMP     AL,1            ; Test caller's preference
        JNE     BEST_MODE_EXIT  ; Go if not preferred
        MOV     DX,5            ; Set mode 5 (direct color)
BEST_MODE_EXIT:
        MOV     AX,DX           ; Result to AX
        POP     DS              ; Restore caller's DS
        RET
XGA_BEST_MODE   ENDP
CODE            ENDS
```

## Setting the XGA Mode

Once the software has determined the available XGA modes and selected the desired one, the mode setting operation takes place. The fundamental mode setting action consists of loading most of the Display Controller registers with preestablished values. These mode-specific values are listed in the *XGA Video Subsystem* section of the *IBM Personal System/2 Hardware Interface Technical Reference Manual*, document number 42G2-2193-00.

The processing operations during mode set are as follows:

1. Disable the video signal to avoid screen garbage during mode setting.
2. Initialize the XGA registers.
3. Clear screen and reenable video.

The following procedure contains the necessary manipulations for setting an XGA mode:

```
;****************************************************************
;       data variables for XGA mode setting operation
;****************************************************************
;
XGA_DIRECT      SEGMENT PUBLIC
;
; Contents of POS registers
XGA_REG_BASE    DW      0FFFFH  ; Register base for XGA system
MODE            DW      0       ; Mode number during init
;
;*********************|
;  mode setting data  |
;*********************|
;    M O D E :      NUMBER:
; 640x480x65536   5 ---------------|
; 640x480x256     4 -----------|   |
; 1024x768x16     3 -------|   |   |
; Code assumes that INIT_XGA has been previously called
; and that an XGA system was found
; On entry:
;         AL = 1 if a direct-color mode is preferred
; On exit:
;         AX holds best available XGA mode
;
;*********************|
;   save caller's DS   |
; set DS to XGA_DIRECT |
;*********************|
        PUSH    DS
        PUSH    AX                  ; Save entry code
        MOV     AX,XGA_DIRECT       ; Local data segment
        MOV     DS,AX               ; to DS
        ASSUME  DS:XGA_DIRECT
        POP     AX                  ; Restore entry code
        MOV     BL,EQUIPMENT        ; Load initialization results
; At this point
; BL bits 7 6 5 4 3 2 1 0
;         | | | | | | | |___ 1 = XGA in system
;         | | | | | | |      0 = no XGA found
;         | | | | | | |_____ 1 = XGA color monitor
;         | | | | | |        0 = XGA monochrome monitor
;         | | | | | |_______ 1 = high-resolution (1024 x 768)
;         | | | | |          0 = no high-resolution
;         | | | | |_________ 1 = RAM = 1Mb
;         | | | |            0 = RAM = 512Kb
;         | | | |___________ 1 = dual monitor system
;         | | |              0 = single monitor system
;         |_|_|_____________ UNUSED
;
;*************************|
;     test for high res     |
;          monitor          |
;*************************|
        MOV     DX,2                ; Assume mode number 2
                                    ; 1024-by-768 pixels in 256
                                    ; colors
        TEST    BL,00000100B        ; High res bit is set?
        JNZ     HIGH_RES_TESTS      ; Go if high res available
; No high-resolution monitor, test for 1Mb RAM
        MOV     DX,5                ; Force mode 5 selection
        TEST    BL,00001000B        ; Bit 3 set if 1Mb
        JNZ     BEST_MODE_EXIT      ; Direct color available
        MOV     DX,4                ; Mode number 4 is best
        JMP     BEST_MODE_EXIT
;*************************|
;    select high res mode   |
;*************************|
; At this point mode number 2 is preselected
```

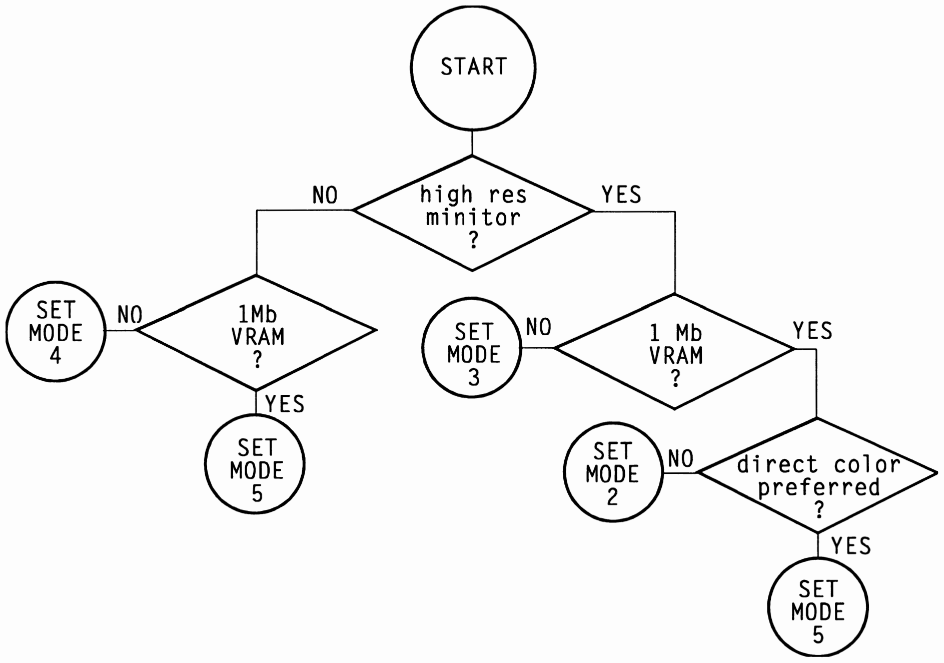


Figure 8.8 *Selection Flowchart for Best XGA Video Mode*

## Selecting the XGA Mode

Since modes number 2 and 5 offer different features, the selection of the best possible XGA mode is a matter of preference and circumstances. For example, mode number 2 (1024-by-768 pixels in 256 colors) is the preferred option when the best resolution is the determining factor, while mode number 5 (640-by-480 pixels in 65,536 colors) is the preferred one when the widest color range is desired. The logic shown in the flowchart of Figure 8.8 attempts to select the best XGA mode, while allowing the user to force direct color output (mode number 5) if modes number 2 and 5 are both available.

The following procedure implements the logic of the flowchart in Figure 8.8 in XGA mode selection:

```
;****************************************************************
;       processing operations for XGA mode selection
;****************************************************************
CODE   SEGMENT PUBLIC
       ASSUME  CS:CODE
.386
;
XGA_BEST_MODE    PROC    FAR
; Select best available XGA mode
```

```
        TEST    EQUIPMENT,10H   ; Test monitor bit for
                                ; dual-monitor system
        JNZ     XGA_EXIT        ; Go if a dual-monitor system
; VGA signal is restored by writing 001B to the XGA Operating
; Mode register and a value of 1 to the VGA Video System Enable
; register
        MOV    DX,XGA_REG_BASE  ; Register base
        MOV    AL,1             ; Bitmap is 001B
        OUT    DX,AL            ; Write to Operating Mode
                                ; register
        MOV    DX,03C3H         ; VGA Video System Enable
                                ; register
        MOV    AL,1             ; Value is 00000001B
        OUT    DX,AL            ; Write to register
        JMP    SHORT XGA_EXIT
;
;**********************|
;      ERROR exit      |
;**********************|
NO_XGA:
        MOV    XGA_REG_BASE,0   ; Set invalid value in variable
        STC                     ; Error flag
        POP     DS              ; Restore caller's DS
        RET
;
;**********************|
;   NO ERROR exit      |
;**********************|
XGA_EXIT:
        MOV     AL,EQUIPMENT    ; Return the result
        MOV     BX,XGA_REG_BASE ; Return register base address
        MOV     CX,MON_ID       ; Monitor ID
        CLC                     ; No error
        POP     DS              ; Restore caller's DS
        RET
;
INIT_XGA        ENDP
CODE            ENDS
```

### 8.4.3 XGA Mode Selection and Setting

Once the XGA preliminary initialization has been performed, the software has available the necessary information to select a particular display mode. The decision is often a simple one since the range of mode options in the XGA is not very extensive at this time. (See Table 8.4.) The graphics mode with the best resolution (mode number 2 in Table 8.4) requires a system with 1Mb of video memory and a monitor capable of supporting 1024-by-768 pixels. The mode with the best color range (mode number 5 in Table 8.4) also requires 1Mb of video RAM and one of the compatible monitors.

```
        MOV     AL,03H            ; 7 6 5 4 3 2 1 0 <== bitmap
                                  ; | | | | | | | |   Bits/pixel
                                  ; | | | | | |_|_|__ 000 = 1 bit
                                  ; | | | | |         001 = 2 bits
                                  ; | | | | |         010 = 4 bits
                                  ; | | | | |        *011 = 8 bits
                                  ; | | | | |         100 = 16 bits
                                  ; | | | | |___  *0 = Intel
                                  ; | | | |        1 = Motorola
                                  ; |_|_|_|____ RESERVED
                                  ; 03H = 00000011B
        OUT     DX,AL
; ... continues in the following code listing
```

## Restore VGA Signal and Exit

At this point the XGA preliminary initialization is complete. All that remains is to reenable video output, to restore the VGA signal if the system is equipped with a single monitor, and to load data into the registers that are passed back to the caller. Recall that the video signal was previously disabled in order to prevent screen garbage during the initialization operations. To disable video, we wrote zeros to the Palette Mask registers; to enable it, we write ones. In a single-monitor system the VGA signal is restored by writing 001B to the XGA Operating Mode register and a value of one to the VGA Video System Enable register.

There are two exit labels: one if no XGA hardware was detected and the other one if an XGA system was found. The code uses the carry flag to report errors (no XGA found if carry set). The information returned to the caller is in the form of a bit-coded equipment byte (shown in the procedure's header), the address of the XGA subsystem (returned in the BX register), and the 4-digit monitor ID code (returned in the CX register). The following fragment shows the processing:

```
; ... continues from the previous code listing
;**********************|
;  enable palette mask |
;**********************|
; Reenable video by setting all bits in the Palette Mask
; register
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
        MOV     AX,0FF64H       ; FFH to Palette Mask register
        OUT     DX,AX
;
;**********************|
; restore VGA signal   |
;**********************|
; In single-monitor systems the VGA signal must be restored
; on exit
; system
        MOV     AL,12            ; Select bank 12
        CALL    XGA_NEWBANK      ; Local select bank procedure
;**********************|
;    test bank 12      |
;**********************|
; Memory size is determined by writing and then reading data at
; bank 12 address. If the data stored is recovered, then the
; XGA system contains 1Kb of VRAM. Otherwise the system contains
; 512K
        PUSH    ES               ; Save ES segment
        MOV     AX,0A000H        ; AX = base address of video RAM
        MOV     ES,AX            ; VRAM base to ES
        MOV     AL,0A5H          ; Any value to AL
        MOV     ES:[0000H],AL    ; Store AL in VRAM
        JMP     SHORT $ + 2      ; Delay
        MOV     AH,ES:[0000H]    ; Read VRAM byte
        CMP     AH,AL            ; Compare values
        JNE     XGA_512          ; Go if value not recovered
; 1Mb VRAM in system
        OR      EQUIPMENT,00001000B     ; Set bit 3
XGA_512:
        POP    ES                ; Restore ES segment
;**********************|
; select video page 0  |
;**********************|
        MOV    AL,0              ; Select bank 0
        CALL   XGA_NEWBANK       ; Restore bank 0 as active
; ... continues in the following code listing
```

## Select Color Depth and Data Format

Before the system microprocessor is able to access video memory, the XGA hardware requires that a specific pixel color depth and data format be selected. This data is entered into the Memory Access Mode register at address 21x9H. The pixel color depth options available are successive powers of 2, that is, 1, 2, 4, 8, and 16 bits per pixel. The data format can be in little endian (Intel convention) or big endian (Motorola convention). PC systems normally use the Intel convention. Although the color depth is mode-specific and could be changed during mode setting, this initialization sets 8 bits-per-pixel and Intel data format in order to leave the XGA in a known state.

```
; ... continues from the previous code listing
;**********************|
;  select access mode  |
;**********************|
; Select Intel order and 8 bits per pixel in the Memory Access
; Mode register (offset + 9)
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,9             ; To Mode register
```

```
; set extended graphics|
;         mode         |
;**********************|
; A value of 100B in bit 0 to 2 of the Operating Mode register
; selects the extended graphics mode
        MOV     DX,XGA_REG_BASE ; Register base
; Switch to extended mode
        MOV     AL,00000100B     ; Bitmap is 100B
        OUT     DX,AL
;**********************|
;  select 64K aperture |
;     at A0000H        |
;**********************|
; Operating systems and real mode applications access XGA video
; memory by means of a 64K aperture. Since in 1024-by-764 mode
; there are 786,432 pixels, 12 banks of 64K are required. The
; active bank is selected by means of the Aperture Index register
; at address 21x8H
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,01H          ; Select Aperture Control
                                ; register
        MOV     AL,1            ; Value 01B selects 64K at A0000H
        OUT     DX,AL           ; Write data to register
; ... continues in the following code listing
```

## XGA Memory size

The recommended way to determine the memory size of an XGA system is by attempting to access memory addresses in the range to be tested. For example, software can determine if the XGA is equipped with 1Mb of video memory by attempting to read and write data to a memory cell located within the last 512K of this range. The code assumes that the 64K aperture is selected.

```
; ... continues from the previous code listing
;**********************|
;    determine XGA     |
;     memory size      |
;**********************|
; Clear palette mask to avoid screen garbage while testing memory
; The contents of the Palette Mask register (offset 64H) are ANDed
; with the screen pixels at display time. Clearing all bits
; disables the display
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,0AH          ; To Index register
        MOV     AX,0064H        ; 00 to register at offset 64H
        OUT     DX,AX           ; Write data
; Display is now disabled by means of the Palette Mask
;**********************|
; select video page 12 |
;**********************|
; All addresses in the page are in the second 512K of a 1Mb
```

```
        OR      AX,BX           ; Store in AX
;***************************|
;     store monitor ID      |
;***************************|
        MOV     MON_ID,AX       ; Variable
; ... continues in the following code listing
```

## Testing for Two-monitor Systems

It is important for software to know if the XGA system is providing the VGA functions or if there is a separately attached VGA monitor. This information can be obtained by testing the low-order bit in the XGA Operating Mode register. If this bit is set, then the XGA is providing the VGA function (single monitor system.) The following code fragment shows the required processing:

```
; ... continues from the previous code listing
;*****************************|
; test for two-monitor system |
;*****************************|
; Read bit 0 of the Operating Mode register (offset 0) to
; determine if VGA mode address decoding is enabled
        MOV     DX,XGA_REG_BASE      ; Operating Mode register
        IN      AL,DX           ; Read data byte
        TEST    AL,1            ; Test low bit
        JNZ     INT_CONTROL     ; Go single monitor in system
        OR      EQUIPMENT,00010000B     ; Set bit 5 to indicate
                                        ; two monitors
; ... continues in the following code listing
```

## Selecting the XGA Aperture

The initialization code must also enable the XGA extended graphics mode and select one of the three possible memory apertures. Notice that software that uses the XGA coprocessor for performing graphics display operations is not linked to any particular aperture. Therefore the aperture is significant only to software that accesses the video memory space directly. The following initialization code fragment selects the XGA 64K aperture:

```
; ... continues from the previous code listing
;*********************|
;  disable XGA system  |
;      interrupts      |
;*********************|
INT_CONTROL:
        MOV     DX,XGA_REG_BASE ; Register base
        ADD     DX,4            ; Select Interrupt Enable
                                ; register
        XOR     AL,AL           ; All interrupts OFF
        OUT     DX,AL
;*********************|
;*************************|
;    merge bits number 0    |
;*************************|
; Now merge all four bits number 0 into monitor nibble 0
        MOV     AL,VAR_A          ; Load 4 byte-registers with bit
        MOV     BL,VAR_B          ; data to be merged
        MOV     CL,VAR_C
        MOV     DL,VAR_D
; Mask out all bits except 0
        AND     AL,00000001B
        AND     BL,00000001B
        AND     CL,00000001B
        AND     DL,00000001B
; Shift registers, except DL
        SHL     AL,1              ; Shift left AL, three times
        SHL     AL,1
        SHL     AL,1
        SHL     BL,1              ; And BL twice
        SHL     BL,1              ; And BL twice
        SHL     CL,1              ; And CL once
; Combine bits in AL and store results
        OR      AL,BL             ; Merge in AL
        OR      AL,CL
        OR      AL,DL
        MOV     NIB_0,AL         ;  Store nibble
;*************************|
;  combine four ID nibbles  |
;      into doubleword      |
;*************************|
; All four nibbles are now combined in the AX register
        MOV     AX,0              ; Clear destination for ORing
        MOV     BX,0              ; And source register
        MOV     BL,NIB_3          ; Load nibble 3
        MOV     CL,12             ; Number of bits to shift
        SHL     BX,CL             ; Shift operand
        OR      AX,BX             ; Store in AX
; Next nibble
        MOV     BX,0              ; And source register
        MOV     BL,NIB_2          ; Load nibble 2
        MOV     CL,8              ; Number of bits to shift
        SHL     BX,CL             ; Shift operand
        OR      AX,BX             ; Store in AX
; Next nibble
        MOV     BX,0              ; And source register
        MOV     BL,NIB_1          ; Load nibble 1
        MOV     CL,4              ; Number of bits to shift
        SHL     BX,CL             ; Shift operand
        OR      AX,BX             ; Store in AX
; last nibble
        MOV     BX,0              ; And source register
        MOV     BL,NIB_0          ; Load nibble 1
        OR      AL,BL           ; Merge in AL
        OR      AL,CL
        OR      AL,DL
        MOV     NIB_3,AL        ;  Store nibble
;***************************|
;    merge bits number 2    |
;***************************|
; Now merge all four bits number 2 into monitor nibble 2
        MOV     AL,VAR_A        ; Load four byte registers with
        MOV     BL,VAR_B        ; bit data to be merged
        MOV     CL,VAR_C
        MOV     DL,VAR_D
; Mask out all bits except 2
        AND     AL,00000100B
        AND     BL,00000100B
        AND     CL,00000100B
        AND     DL,00000100B
; Shift registers, except BL
        SHL     AL,1            ; Shift left AL
        SHR     CL,1            ; Shift right CL bits
        SHR     DL,1            ; Shift right DL bits twice
        SHR     DL,1
; Combine bits in AL and store results
        OR      AL,BL           ; Merge in AL
        OR      AL,CL
        OR      AL,DL
        MOV     NIB_2,AL       ;  Store nibble
;
;***************************|
;    merge bits number 1    |
;***************************|
; Now merge all four bits number 1 into monitor nibble 1
        MOV     AL,VAR_A        ; Load four byte registers with
        MOV     BL,VAR_B        ; bit data to be merged
        MOV     CL,VAR_C
        MOV     DL,VAR_D
; Mask out all bits except 1
        AND     AL,00000010B
        AND     BL,00000010B
        AND     CL,00000010B
        AND     DL,00000010B
; Shift registers, except CL
        SHL     AL,1            ; Shift left AL, twice
        SHL     AL,1
        SHL     BL,1            ; And BL once
        SHR     DL,1            ; Shift DL bits once
; Combine bits in AL and store results
        OR      AL,BL           ; Merge in AL
        OR      AL,CL
        OR      AL,DL
        MOV     NIB_1,AL        ;  Store nibble
        MOV     VAR_B,AL         ; Store in variable
;***************************|
;     read data nibble C    |
;***************************|
; SP field = 00 for this read cycle
        CALL    READ_DC1
        AND     AH,00111111B     ; Clear bits 6 and 7
        MOV     AL,50H           ; Register number in AL
        OUT     DX,AX            ; Write to port in DX
; Read Display ID and comparator register at offset 52H
; using local procedure
        CALL    READ_DID         ; Local procedure
; AL has bits read
        AND     AL,00001111B     ; Clear high nibble
        MOV     VAR_C,AL         ; Store in variable
;***************************|
;     read data nibble D    |
;***************************|
; SP field = 11 for this read cycle
        CALL    READ_DC1
        AND     AH,00111111B     ; Clear bits 6 and 7
        OR      AH,11000000B
        MOV     AL,50H           ; Register number in AL
        OUT     DX,AX            ; Write to port in DX
; Read Display ID and comparator register at offset 52H
; using local procedure
        CALL    READ_DID         ; Local procedure
; AL has bits read
        AND     AL,00001111B     ; Clear high nibble
        MOV     VAR_D,AL         ; Store in variable
;***************************|
;       merge ID data       |
;***************************|
; First merge all four bits number 3 into monitor nibble 3
        MOV     AL,VAR_A         ; Load four byte registers with
        MOV     BL,VAR_B         ; bit data to be merged
        MOV     CL,VAR_C
        MOV     DL,VAR_D
; Mask out all bits except 3
        AND     AL,00001000B
        AND     BL,00001000B
        AND     CL,00001000B
        AND     DL,00001000B
; Shift registers, except AL
        SHR     BL,1             ; Shift right and merge
        SHR     CL,1             ; Shift CL bits twice
        SHR     CL,1
        SHR     DL,1             ; Shift DL bits three times
        SHR     DL,1
        SHR     DL,1
; Combine bits in AL and store results
; Controller 1 register (offset 50H)
        CALL    READ_DC1        ; Get byte in Display Cont. 1
                                ; using local procedure
; Al now holds contents of Display Control 1 register
; Display Blanking bit field (bits 0-1) must be set to 01
; to prepare for reset. Bit 2 must also be set
        OR      AL,00000101B    ; Set bits 0 and 2
        AND     AL,11111101B    ; Clear bit 1
        MOV     DX,XGA_REG_BASE ; Base to DX
        ADD     DX,0AH          ; To Index register
        MOV     AH,AL           ; Data byte to AH
        MOV     AL,50H          ; Port number to AL
        OUT     DX,AX
; Display Blanking bit field (bits 0-1) must be set to 00
; to reset. Bit 2 must also be set
        CALL    READ_DC1
        OR      AH,00000100B    ; Set bit 2
        AND     AH,11111100B    ; Clear bits 0 and 1
        MOV     AL,50H          ; Port number to AL
        OUT     DX,AX           ; Write to port in DX
; Set Sync Polarity field (bits 6 -7) to 01. This sets the
; VSYNC to 0 and HSYNC to 1
        CALL    READ_DC1
        AND     AH,00111111B    ; Clear bits 6 and 7
        OR      AH,01000000B    ; Set bit 6
        MOV     AL,50H          ; Register number in AL
        OUT     DX,AX           ; Write to port in DX
;***************************|
;     read data nibble A    |
;***************************|
; SP field = 01 for first read
; Read Display ID and Comparator register at offset 52H
; using local procedure
        CALL    READ_DID        ; Local procedure
; AL has bits read
        AND     AL,00001111B    ; Clear high nibble
        MOV     VAR_A,AL        ; Store in variable
;***************************|
;     read data nibble B    |
;***************************|
; SP field = 10 for this read cycle
        CALL    READ_DC1        ; Local procedure
        AND     AH,00111111B    ; Clear bits 6 and 7
        OR      AH,10000000B
        MOV     AL,50H          ; Register number in AL
        OUT     DX,AX           ; Write to port in DX
; Read Display ID and Comparator register at offset 52H
; using local procedure
        CALL    READ_DID        ; Local procedure
; AL has bits read
        AND     AL,00001111B    ; Clear high nibble
```

Chapter

10

# SuperVGA Graphics and Animation

## 10.0 SuperVGA

The name SuperVGA refers to enhancements to the VGA standard usually in the form of video adapters designed for computers based on the ISA or EISA bus architectures. One common characteristic of all SuperVGA boards is the presence of graphics features that exceed the VGA standard in definition or color range. In other words, a typical SuperVGA board is capable of executing not only the standard VGA modes, but also other modes that provide higher definition or more colors than VGA. The proprietary SuperVGA modes are usually called the *enhanced modes*.

The proliferation of SuperVGA hardware during the late eighties gave rise to many compatibility problems. This was due to the fact that the enhanced features of the SuperVGA cards were not standardized; therefore the SuperVGA enhancements in the card produced by one manufacturer were often incompatible with the enhancements in a card made by another company. To the graphics programmer this situation often presented insurmountable problems, because a program designed to take advantage of the enhancements in one SuperVGA card would usually not execute correctly in another one.

These incompatibility problems are easier to correct at the operating system level than at the application level, particularly regarding operating systems that offer graphics services to application software. For example, the manufacturer of SuperVGA boards can furnish software drivers for Windows and OS/2. Once the driver is installed, the graphics environment in the operating system is able to use the enhancements provided by a particular SuperVGA board and provide these services to applications that request them.

In addition, the MS-DOS version of some high-end graphics programs have been designed with a flexible video interface to make them more easily adapt to the features of a particular SuperVGA. Some versions of AutoCad, Ventura Publisher, Wordperfect, Lotus 1-2-3, and others have video hooks to which a SuperVGA card can attach its low-level driver. Therefore a SuperVGA manufacturer can make available a driver program to make its hardware compatible with a particular application.

However, in many high-performance graphics applications the video functions are embedded in the code. In this case the adaptation to a nonstandard video mode or hardware usually implies a major program redesign. In 1989 several manufacturers of SuperVGA boards formed the *Video Electronics Standards Association* (VESA) in an attempt to solve this lack of standardization. In October 1989 VESA released its first SuperVGA standard, which defined several enhanced video modes and implemented a BIOS extension designed to provide a few fundamental video services in a compatible fashion.

### 10.0.1 SuperVGA in Animation Programming

The problem of SuperVGA programming boils down to two options: the SuperVGA system can be accessed at the hardware level, with maximum performance but minimum portability to other SuperVGAs, or the system can be accessed by means of a software interface, such as that provided by the VESA standard, which assures portability at the expenses of performance. To the animation programmer these options offer no easy choice. Developing software that runs only in one particular hardware configuration is usually not a commercially viable alternative. On the other hand, the performance sacrifice that is required to achieve program portability is not always technically possible.

## 10.1 SuperVGA Memory Architecture

All IBM microcomputer video systems are memory-mapped. VGA video memory extends from A0000H to BFFFFH. The 64K block from A0000H to AFFFFH is usually devoted to graphics while the 64K block from B0000H to BFFFFH is for alphanumeric modes. The total space reserved for video operations is 128K. However, since some systems are set up with two monitors, one of them operating in alphanumeric modes (base address B000H), the actual video space for graphics operations is practically limited to 64K.

The video data that can be stored in a 64K memory space is limited. In VGA mode number 19, in order to achieve 256 colors the screen definition must be reduced to 320-by-200 pixels. VGA mode X designers were able to increase this definition to 320-by-240 by introducing a planar mechanism that stores data in a similar structure as VGA mode 18. But in spite of the short supply of video memory space, simple arithmetic shows a memory surplus in many VGA modes. For example, if the resolution is of 640-by-480 pixels (mode 18), the video data stored in each map takes up 38,400 bytes of the available 65,536. Therefore, there are 27,136 unused bytes in each map. The idea of enhancing the VGA system was based on using this surplus memory to store video data.

The original SuperVGA designers realized that it was possible to have an 800-by-600 pixel display, divided into four maps of 60,000 bytes each, and yet not exceed the 64K space allowed for each color map nor the total 265K furnished with the VGA system. Enhancing the 16-color VGA modes to a resolution of 800-by-600 pixels was one of the first extensions to the VGA standard. This mode, which was later designated as mode 6AH by the VESA standard, could be programmed in a similar manner as VGA mode 18. The VGA extension, which could be achieved with minor changes in the hardware, provided a 36 percent increase in the display area.

Another apparently expedient extension to the VGA standard can be achieved by means of a wider pixel mask register. This change in the hardware would make possible the use of more than 16 colors in the corresponding VGA modes. Every bit added to the pixel mask would double the color range. However, this has never been implemented in a SuperVGA system due to performance factors and other hardware considerations.

## 10.1.1 SuperVGA Memory Banking

The memory structure for VGA mode number 19, in 256 colors, is based, not on a multiplane layout, but on a much simpler scheme that maps a memory byte to each screen pixel. (See Figure 2.12.) In this manner, 256 color combinations can be directly encoded into a data byte, which conveniently corresponds to the 256 DAC color registers of the VGA hardware. The method appears straightforward and uncomplicated; however, if the entire video space is to be contained in 64K of memory, the maximum resolution would be limited to 65,535 pixels. In other words, a rectangular screen of 320-by-200 pixels, such as the one used in VGA mode number 19, nearly fills the allotted 64K. Figure 10.1 shows mapping of several memory banks to the video display.

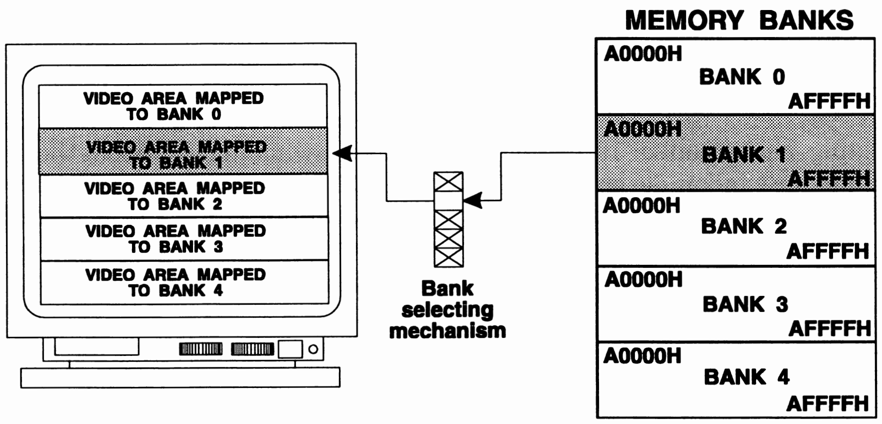

MEMORY BANKS
A0000H
BANK 0
AFFFFH
A0000H
BANK 1
AFFFFH
A0000H
BANK 2
AFFFFH
A0000H
BANK 3
AFFFFH
A0000H
BANK 4
AFFFFH
VIDEO AREA MAPPED TO BANK 0
VIDEO AREA MAPPED TO BANK 1
VIDEO AREA MAPPED TO BANK 2
VIDEO AREA MAPPED TO BANK 3
VIDEO AREA MAPPED TO BANK 4
Bank selecting mechanism

Figure 10.1 *SuperVGA Memory Banking*

Therefore, if the resolution for a 256-color mode were to exceed 64K pixels, it would be necessary to find other ways of mapping video memory into 64K of system RAM. One such alternative is the VGA mode X described in Chapter 7. The mechanism adopted by the SuperVGA designers was based on the well-known technique known as bank switching. In bank switching the video display hardware maps several 64K blocks of RAM to the same video memory area. Addressing of the multisegment RAM space is by means of a hardware mechanism that selects which video memory area is currently located at the system's addressable space. In the SuperVGA implementation the system addressable space is usually at A0000H. The entire process is reminiscent of memory page switching in the *Lotus / Intel / Microsoft* (LIM ) Extended Memory environment.

The term *aperture* is often used in video graphics terminology to denote the processor's window into the video memory space. For example, if the addressable area of video memory starts at physical address A0000H and extends to AFFFFH, we say that the CPU has a 64K aperture into video memory (10000H = 64K). In SuperVGA documentation the word "granularity" is often used in this context. In Figure 10.1 we can see that the bank selector determines which area of video memory is mapped to the processor's aperture, the same video display area that can be updated by the processor. In other words, in a memory banking scheme the processor cannot access the entire video memory at once. Therefore, in the case shown in Figure 10.1 we would have to perform five bank switches in order to update the entire screen. By the same token, if the code intends to update a pixel located in bank number 0, it must first activate the bank selection mechanism so that bank number 0 is active.

### 10.1.2 SuperVGA 256-Color Extensions

The 256-color SuperVGA alternative is often based on a banking mechanism similar to the one shown in Figure 10.1. The usual scheme is to use a memory byte to encode the 256 color combinations for each screen pixel, and to do away with the pixel masking complications of VGA mode number 18. This method is characteristic of the SuperVGA extensions and has no precedent in CGA, EGA, or VGA systems. Although it is similar to VGA mode number 19 regarding color encoding, mode number 19 does not require bank switching. Note that the neat, rectangular window design shown in Figure 10.1 does not always conform with reality. Several implementations of SuperVGA multicolor modes use nonrectangular windows that start and end inside a screen scan line.

The total memory installed in a SuperVGA system determines the available resolution and color range. For example, we have seen that a 800-by-600 pixel mode can be implemented in 16 colors in a system with no more than 256K. However, if the color range were to be 256 colors, requiring one memory byte per screen pixel, the SuperVGA system would need 480,000 bytes. By the same token, a resolution of 1024-by-768 pixels in 256 colors requires 786,432 bytes.

### 10.1.3 SuperVGA Pixel Addressing

The calculations for setting an individual pixel in the 256-color SuperVGA modes depend upon the size of the memory banks, the number of pixels per row, the number of screen rows, and the start address of video memory. Although it is quite feasible to design a routine that performs in different SuperVGA chipsets, the efficiency of such code would be necessarily low. The VESA standardization offers a solution to the programming complications brought on by different architectures of the various SuperVGA chipsets. In reality, since most SuperVGA systems use a 64K bank size and a processor's window into video memory located at address A0000H, the variations are reduced to the bank switching operations.

## 10.2 SuperVGA Architecture

In 1989, in an attempt to solve the portability problems created by the proliferation of nonstandard VGA hardware, several manufacturers of so-called SuperVGA boards formed the Video Electronics Standards Association (VESA). At present over 150 companies are members of this organization. In October of 1989 VESA released its first SuperVGA standard. The VESA standard defined several enhanced video modes and implemented a BIOS extension designed to provide a few fundamental video services in compatible fashion. Because of this advantage in compatibility and portability, our treatment of SuperVGA programming focuses on the use of the VESA BIOS functions. As previously mentioned, this convenience comes at a performance price.

### 10.2.1 The VESA SuperVGA Standard

The Video Electronics Standards Association was created for the purpose of providing a common programming interface for SuperVGA extended modes. In order to achieve this, each manufacturer furnishes a VESA SuperVGA BIOS extension. The BIOS can be in the adapter ROM or in a TSR routine.

The first release of the VESA SuperVGA standard was published October 1, 1989 (version 1.0). A second release was published in June 2, 1990 (version 1.1). The present release is dated October 22, 1991 (version 1.2).

### 10.2.2 VESA SuperVGA Modes

The first element of VESA standardization is the definition of standard modes for the SuperVGA extensions. The VESA mode numbering scheme takes into account that encoding for the VGA modes extends to the value FFH due to the fact that the VGA BIOS mode setting function (service number 0) uses the high-order bit to determine if video memory is to be cleared. To get around this restriction, the VESA mode number is a word-size value passed to the VESA BIOS in the BX register. Figure 10.2 shows the bitmap of the VESA MODE numbers.

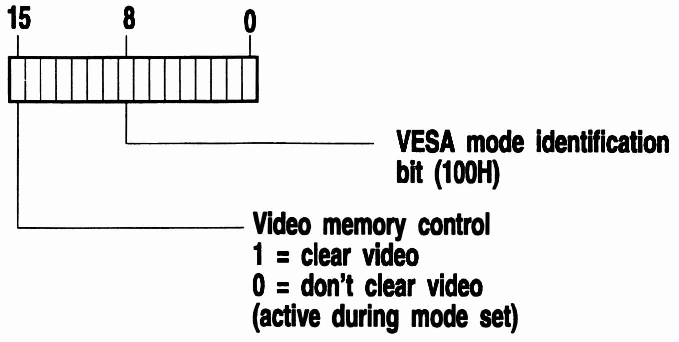


Figure 10.2 *VESA Mode Numbering Bitmap*

Notice in Figure 10.2 that bit number 8 identifies a VESA mode. Therefore, all VESA modes start at number 100H. Also notice that bit number 15 is used during mode set operations to indicate if video memory is to be cleared. Table 10.1 lists the VESA extended modes.

Table 10.1 *VESA BIOS Modes*

| MODE NUMBER 15 BITS | MODE NUMBER 7 BITS | TEXT/ GRAPHICS | PIXELS | RESOLUTION COLUMNS/ROWS | COLORS |
|---|---|---|---|---|---|
| 100H | | GRAPHICS | 640-by-400 | | 256 |
| 101H | | GRAPHICS | 640-by-480 | | 256 |
| 102H | 6AH | GRAPHICS | 800-by-600 | | 16 |
| 103H | | GRAPHICS | 800-by-600 | | 256 |
| 104H | | GRAPHICS | 1024-by-768 | | 16 |
| 105H | | GRAPHICS | 1024-by-768 | | 256 |
| 106H | | GRAPHICS | 1280-by-1024 | | 16 |
| 107H | | GRAPHICS | 1280-by-1024 | | 256 |
| 108H | | TEXT | | 80-by-60 | |
| 109H | | TEXT | | 132-by-25 | |
| 10AH | | TEXT | | 132-by-43 | |
| 10BH | | TEXT | | 132-by-50 | |
| 10CH | | TEXT | | 132-by-60 | |
| * 10DH | | GRAPHICS | 300-by-200 | | 32K |
| 10EH | | GRAPHICS | 320-by-200 | | 64K |
| 10FH | | GRAPHICS | 320-by-200 | | 16.8Mb |
| 110H | | GRAPHICS | 640-by-480 | | 32K |
| 111H | | GRAPHICS | 640-by-480 | | 64K |
| 112H | | GRAPHICS | 640-by-480 | | 16.8Mb |
| 113H | | GRAPHICS | 800-by-600 | | 32K |
| 114H | | GRAPHICS | 800-by-600 | | 64K |
| 115H | | GRAPHICS | 800-by-600 | | 16.8Mb |
| 116H | | GRAPHICS | 1024-by-768 | | 32K |
| 117H | | GRAPHICS | 1024-by-768 | | 64K |
| 118H | | GRAPHICS | 1024-by-768 | | 16.8Mb |
| 119H | | GRAPHICS | 1280-by-1024 | | 32K |
| 11AH | | GRAPHICS | 1280-by-1024 | | 64K |
| 11BH | | GRAPHICS | 1280-by-1024 | | 16.8Mb |

* modes after 10DH were introduced in VESA BIOS version 1.2

## 10.2.3 Memory Windows

The VESA standard accommodates variations in the SuperVGA implementations by recognizing two different types of hardware windows into video memory. The first and simpler type consists of a single window which can be read and written by the CPU. The disadvantage of a read-write window becomes evident when a pixBlt operation crosses the limit of the window. In this case, the software is forced to switch banks and the CPU is forced to reset the segment register base during the transfer. This double burden can considerably degrade performance.

A partial solution is to provide separate windows for read and write operations. One possible option is to have two windows located at the same address: one for read and the other one for write operations. This scheme, sometimes called *dual overlapping windows*, allows selecting both windows simultaneously. Once the source and destination windows are selected, the data block can be rapidly moved by means of a REP MOVSB instruction.

A second alternative to the two windows option is to locate the read and write windows at separate addresses. For example, the write window can be located at base address A000H and the read window at B000H. This would extend addressable memory to 128K and considerably simplify pixBlt operations. The objection to this approach is that a two-monitor system requires the B000H window for text operations; therefore this configuration would not be possible.

A third solution is to cut the 64K window in half and provide two separate 32K windows, one for read and the other one for write operations. The objection in this case is that normal display operations would require twice as many bank switches. Figure 10.3 is a schematic representation of the three possible windowing options.

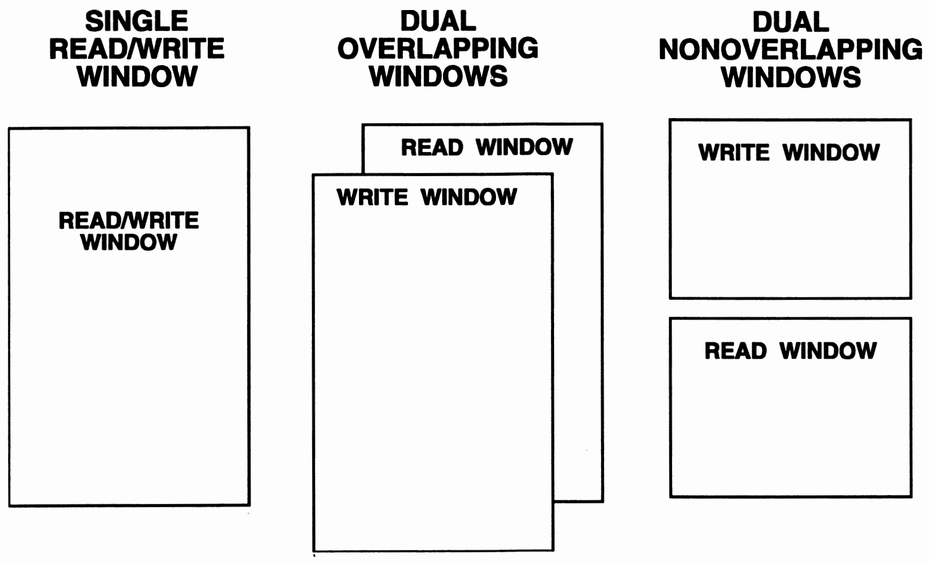


Figure 10.3 *Windowing Options in the VESA Standard*

## 10.3 The VESA BIOS

The VESA BIOS has been designed to perform only those operations that are strictly necessary to achieve portability and hardware transparency of the SuperVGA system. The fundamental functions of the VESA BIOS, as used in SuperVGA programming, are the following:

1. Obtaining SuperVGA and mode information
2. Setting a standard VESA extended mode
3. Performing bank switching operations

The VESA BIOS does not provide graphics primitives. Furthermore, not even pixel setting and reading operations are included in the standard. Due to this design the software overhead is kept at a minimum. The actual implementations of the functions are left to the chipset manufacturer.

Of the functions provided by the VESA BIOS, the bank switching operation is the most crucial regarding display system performance, because bank switching is usually included in read and write loops and, therefore, is in the program's critical path of execution. To provide the best possible performance, the VESA BIOS allows access to the bank switching function directly, by means of a far call to the chipset manufacturer's own entry point to the service routine. This approach simplifies and accelerates access to the actual bank switching code. The result is that display routines that use VESA BIOS functions can perform bank switching operations almost as efficiently as routines that access the SuperVGA hardware directly.

The VESA BIOS is an extension of VGA BIOS video services located at interrupt 10H. Access to the VESA BIOS is by means of service number 79 (4FH). The subfunction refers to the specific VESA BIOS service. Eight VESA BIOS services have been implemented to date. These are shown in Table 10.2.

Table 10.2 *SuperVGA BIOS Extension BIOS INT 10H*

| SUBSERVICE | DESCRIPTION |
|---|---|
| 00H | Return SuperVGA information |
| 01H | Return SuperVGA mode information |
| 02H | Set SuperVGA mode |
| 03H | Return current video mode |
| 04H | Save/restore SuperVGA video state |
| 05H | Switch banks |
| 06H | Set/get logical scan line length |
| 07H | Set/get display start |

The following code fragment is a general template for accessing the VESA BIOS subservices:

```
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,??           ; AL holds subservice number
         .                      ; Other registers are loaded with
         .                      ; the values required by the
         .                      ; subservice
        INT     10H
```

All VESA BIOS functions return the same error codes: AL = 79 (4FH) if the function is supported, and AH = 0 if the call was successful.

### 10.3.1 Subservice 0 — System Information

VESA BIOS subservice number 0 provides general VESA information. The caller furnishes a pointer to a 256-byte data buffer which is filled by the VESA service. The following procedure shows the processing required for calling this VESA BIOS service:

```
;****************************************************************
;            data structure for VESA information
;****************************************************************
;
SVGA_DATA        SEGMENT
;
;**********************|
;    VESA information  |
;**********************|
VESA_BUFFER     DB      '    '          ; VESA signature
VESA_VERSION    DW      ?       ; Version number
OEM_PTR_OFF     DW      ?       ; OEM string offset pointer
OEM_PTR_SEG     DW      ?       ; OEM string segment pointer
CAPABILITIES    DB      4 DUP (00H)  ; Reserved field
MODES_PTR_OFF   DW      ?       ; Pointer to modes list, offset
MODES_PTR_SEG   DW      ?       ; Segment for idem
MEM_BLOCKS      DW      ?       ; Count of 64K memory blocks
                                ; (Only in June 2, 1990 revision)
                DB      242 DUP (0H)
;
;**********************|
;  first field group   |
;**********************|
VESA_DATA       DW      ?       ; Mode attributes, mapped as
                        ; follows:
                        ; ..4 3 2 1 0 <= bits
                        ;   | | | | |__ 0 = mode not supported
                        ;   | | | |     1 = mode supported
                        ;   | | | |____ 0 = no extended mode info
                        ;   | | |       1 = extended mode info
                        ;   | | |_______0 = no output functions
                        ;   | |         1 = output functions
                        ;   | |________ 0 = monochrome mode
                        ;   |           1 = color mode
                        ;   |__________ 0 = text mode
                        ;               1 = graphics mode
                        ; 15..5 = RESERVED
WIN_A_ATTS      DB      ?       ; Window A attributes
WIN_B_ATTS      DB      ?       ; Window B attributes
WIN_GRAIN       DW      ?       ; Window granularity
```

```
WIN_SIZE        DW      ?       ; Window size
WIN_A_SEG       DW      ?       ; Segment address for window A
WIN_B_SEG       DW      ?       ; Segment address for window B
SWITCH_BANK     DD      ?       ; Far pointer to bank switch
                                ; function
BYTES_PER_ROW   DW      ?       ; Bytes per screen row
;
;*********************|
;  second field group |
;*********************|
; Extended mode data. Optional until VESA BIOS version 1.2
X_RES           DW      ?       ; Horizontal resolution
Y_RES           DW      ?       ; Vertical resolution
X_CHAR_SIZE     DB      ?       ; Pixel width of character cell
Y_CHAR_SIZE     DB      ?       ; Pixel height of character cell
BIT_PLANES      DB      ?       ; Number of bit planes
BITS_PER_PIX    DB      ?       ; Bits per pixel in this mode
NUM_OF_BANKS    DB      ?       ; Number of video memory banks
MEM_MODEL       DB      ?       ; Memory model, as follows:
                                ; 00H = text mode
                                ; 01H = CGA graphics
                                ; 02H = Hercules graphics
                                ; 03H = 4-plane architecture
                                ; 04H = Packed pixel architecture
                                ; 05H = 256 color (unchained)
                                ; The following were defined
                                ; in VESA BIOS version 1.2:
                                ; 06H = Direct color
                                ; 07H = YUV color
                                ; 08H - 0FF = not yet defined
BANK_SIZE       DB      ?       ; Kilobytes per bank
PLANES          DB      ?       ; Number of planes:
                                ; 4 in 16 color modes
                                ; 1 in 256 color modes
                DB       1      ; Reserved for BIOS
;
;*********************|
;  third field group  |
;*********************|
; Direct color fields. Defined in VESA BIOS version 1.2
RED_MASK        DB      ?       ; Bit size of red mask
RED_POSITION    DB      ?       ; Red mask LSB position
GREEN_MASK      DB      ?       ; Bit size of green mask
GREEN_POSITION  DB      ?       ; Green mask LSB position
BLUE_MASK       DB      ?       ; Bit size of blue mask
BLUE_POSITION   DB      ?       ; Blue mask LSB position
RSVD_MASK       DB      ?       ; Bit size of reserved mask
RSVD_POSITION   DB      ?       ; Reserved mask LSB position
DC_INFO         DB      ?       ; Attributes of direct color
                                ; modes, as follows:
                                ; bit 0 = color ramp
                                ;       0 = fixed
                                ;       1 = programmable
                                ; bit 1 = Reserved field bits
                                ;       0 = not usable
                                ;       1 = usable
                DB      216 DUP (?) ; Remainder of block
SVGA_DATA        ENDS
;
;****************************************************************
;                           procedures
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
VESA_INFO       PROC    FAR
; Call VESA BIOS subservice number 0 to obtain SuperVGA
; information and subservice number 1 to obtain mode information
; On entry:
;          CX = mode number, as follows:
;               number       resolution      colors
;                100H         640-by-400       256
;                101H         640-by-480       256
;                102H         800-by-600        16
;                103H         800-by-600       256
;                104H        1024-by-768        16
;                105H        1024-by-768       256
;                106H        1280-by-1224       16
;                107H        1280-by-1224      256
; On exit:
;         Carry clear if no error
;         Data stored in the buffers VESA_BUFFER and VESA_DATA
;         ES:SI -> VESA_BUFFER
;         ES:DI -> VESA_DATA
;
;*********************|
;   save caller's DS  |
; set DS to SVGA_DATA |
;*********************|
        PUSH    DS
        MOV     AX,SVGA_DATA            ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:SVGA_DATA
;
;*********************|
;   setup registers   |
;*********************|
; Set pointers to data storage buffers
        LEA     DI,VESA_BUFFER  ; Pointer to data buffer
        LEA     SI,VESA_DATA
; VESA BIOS subservice number 0 uses ES as a segment base
        PUSH    DS              ; Local data segment
        POP     ES                ; To ES
;
;**********************|
; get VESA information |
;**********************|
        MOV     AH,79             ; VESA BIOS service number
        MOV     AL,0              ; This subservice
        INT     10H               ; BIOS video service
; At this point AX must hold 004FH if the call executed
        CMP     AX,004FH          ; Returned code
        JNE     BAD_VESA          ; Go if invalid value
; Test buffer for a valid 'VESA' signature
        CMP     WORD PTR [DI],'EV'        ; First two letters
        JE      OK_VE             ; Go if matched
        JMP     BAD_VESA          ; Exit if not matched
OK_VE:
        CMP     WORD PTR [DI+2],'AS'      ; Last two letters
        JE      OK_VESA           ; Go if signature matched
        JMP     BAD_VESA          ; Go if not matched
;**********************|
;  get VESA mode info  |
;**********************|
OK_VESA:
; At this point there is a valid call and signature
; CX holds requested mode number
; VESA BIOS subservice number 1 is used to obtain mode
; information
        XCHG    SI,DI             ; Set DI as pointer
        MOV     AH,79             ; VESA BIOS service number
        MOV     AL,1              ; This subservice
        INT     10H               ; BIOS video service
;
        POP     DS                ; Restore caller's DS
        XCHG    SI,DI             ; Pointers to original registers
        CLC
        RET
BAD_VESA:
; Error exit
        POP     DS                ; Restore caller's DS
        STC                       ; Error flag
        RET
VESA_INFO       ENDP
P_CODE          ENDS
```

The call to subservice number 0 is usually made to determine if there is a VESA BIOS available, although the subservice provides other information that could also be useful. Testing for a valid VESA BIOS is a two-step process: first the code tests for the value 004FH in the AX register. This value corresponds to the standard VESA error codes mentioned at the beginning of this section. Once this first test is passed, the code makes certain that the 4-character 'VESA' signature is stored at the start of the buffer. If these tests are satisfactory, execution can continue on the assumption that a valid VESA BIOS is present and that its functions are available to the software.

The data segment of the VESA_INFO procedure shows the most important items returned by subservice number 0. The field contents are as follows:

VESA_BUFFER is the label that marks the start of the buffer. At this label the BIOS stores the word 'VESA' which serves as a string signature that identifies the BIOS.

VESA_VERSION is a two-byte field that encodes the current version of the VESA BIOS. The encoding is in fractional form; for example, the value 3131H corresponds to the ASCII digits 1,1 and represents version 1.1 of the VESA BIOS. An application can assume upward compatibility in the VESA BIOS.

OEM_PTR_OFF and OEM_PTR_SEG are two word variables that encode the offset and segment values of a far pointer to an identification string supplied by the board manufacturer. Board-specific routines would use this string to check for compatible hardware.

The CAPABILITIES label is a 4-byte field designed to hold a code that represents the general features of the SuperVGA environment. This field was not used until VESA BIOS version 1.2, released on October 22, 1991. At this time bit number 0 of this field was enabled to encode adapters with the possibility of storing extended primary color codes. In VESA BIOS version 1.2, and later, a value of 1 in bit 0 of the CAPABILITIES field indicates that the DAC registers can be programmed to hold more than 6-bit color codes. A value of 0 indicates that the DAC register is standard VGA, with 6-bits per primary color. Changing the bit width of the DAC registers is performed by calling subservice number 8, discussed later in this section.

MODES_PTR_OFF and MODES_PTR_SEG are two word variables that hold the offset and segment values of a far pointer to a list of implemented SuperVGA modes. Each mode occupies one word in the list. The code 0FFFFH serves as a list terminator. An application can examine the list of modes to make certain that a specific one is available or to select the best one among possible candidates.

The MEM_BLOCKS field encodes, in a word variable, the number of 64K blocks of memory installed in the adapter. Note that this field was first implemented in VESA BIOS version 1.1.

### 10.3.2 Subservice 1 — Mode Information

VESA BIOS subservice number 1 provides information about a specific SuperVGA VESA mode. The caller furnishes a pointer to a 256-byte data buffer, which is filled by the VESA service, as well as the number of the desired mode.

The call to subservice number 1 is usually made to determine if the desired mode is available in the hardware and, if so, to obtain fundamental parameters required by the program. If the call is successful, the code can examine the data at offset 0 in the data buffer in order to determine the mode's fundamental attributes. These mode attributes are shown in Figure 10.4.

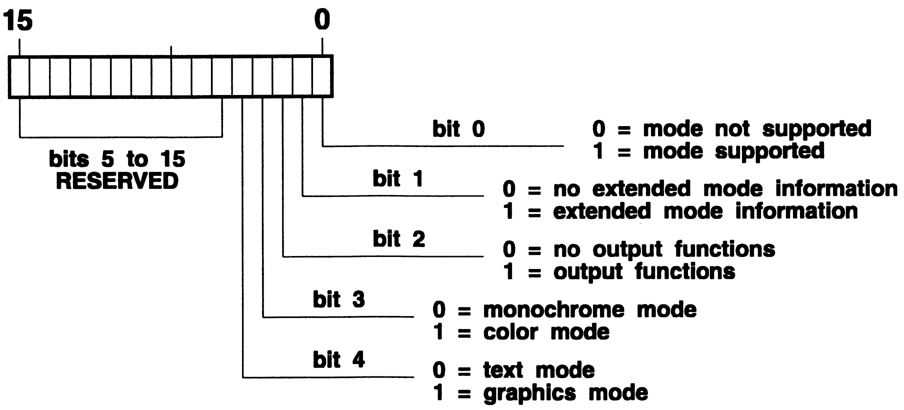


Figure 10.4 *VESA Mode Attributes Bitmap*

The data segment of the procedure named VESA_INFO shows the items returned by subservice number 1. The data items are divided into three field groups. The contents of the variables in the first field group are as follows:

WIN_A_ATTS and WIN_B_ATTS are two bytes that encode the attributes of the two possible memory banks, or windows. Figure 10.5 is a bitmap of the window attribute bytes. The code can inspect the window attribute bits to determine the window types used in the system (see Figure 10.3).

The WIN_GRAIN word specifies the granularity of each window. The granularity unit is one kilobyte. The value can be used to determine the minimum video memory boundary for the window.

The WIN_SIZE word specifies the size of the windows in kilobytes. This value can be used in tailoring bank switching operations to specific hardware configurations.

The word labeled WIN_A_SEG holds the segment base address for window A and the word labeled WIN_B_SEG holds the base address for window B. The base address in graphics modes is usually A000H; however, the code should not take this for granted.

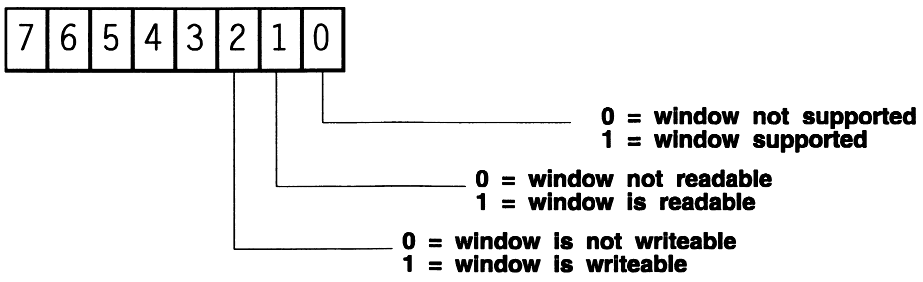


Figure 10.5 *VESA Windows Attributes Bitmap*

The doubleword labeled BANK_FUN holds a far pointer to the bank shifting function in the BIOS. An application can shift memory banks using VESA BIOS subservice number 5, described later in this section, or by means of a direct call to the service routine located at the address stored in this variable. The call can be coded with the instruction:

```
        CALL    DWORD PTR BANK_FUN
```

BYTES_PER_ROW is a word variable that encodes the number of bytes in each screen logical pixel row. Note that this value can be larger than the number of pixels in a physical scan line.

The variables in the second field group are of an optional nature. Bit number 1 of the mode attribute bitmap (see Figure 8.4) can be read to determine if this part of the data block is available. The contents of the various fields in the second group are described in the data segment of the preceding code fragment.

The direct color fields form the third field group. These fields were first implemented in VESA BIOS version 1.2 to support SuperVGA systems with color capabilities that extend beyond the 256 color modes. The contents of the various fields in the third group are described in the data segment of the preceding code fragment.

### 10.3.3 Subservice 2 — Set Video Mode

VESA BIOS subservice number 2 is used to initialize a video mode supported by the adapter. The VESA mode number is passed to the subservice in the BX register. The high-order bit, sometimes called the *clear memory flag*, is set to request that video memory not be cleared. The following procedure shows the processing operations for setting VESA BIOS mode number 105H:

```
;****************************************************************
;    processing operations for setting VESA BIOS mode 105H
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
VESA_MODE_105   PROC    FAR
; Procedure to set SuperVGA mode number 105H with a resolution
; of 1024-by-768 pixels in 256 colors
; This procedure assumes that the data variables in the buffers
; VESA_BUFFER and VESA_DATA have been filled by a previous call
; to the VESA_INFO procedure with CX = 105H
;**********************|
;   save caller's DS   |
; set DS to SVGA_DATA  |
;**********************|
        PUSH    DS
        MOV     AX,SVGA_DATA            ; Local data segment
        MOV     DS,AX                   ; to DS
        ASSUME  DS:SVGA_DATA
```

```
;**********************|
; test standard values |
;**********************|
        MOV     AX,WIN_SIZE     ; Standard size is 64K (40H)
        CMP     AX,040H         ; Is it 64K
        JE      OK_WIN_SIZE     ; Go if standard size
; Error exit. Nonstandard parameters
;**********************|
;     ERROR exit       |
;**********************|
NON_STANDARD:
        POP     DS              ; Restore caller's DS
        STC                     ; Carry is error flag
        RET
; Now test for a start address of video buffer at A000H
OK_WIN_SIZE:
        MOV     AX,WIN_A_SEG    ; Segment for window A
        CMP     AX,0A000H       ; Test for standard address
        JE      SET_MODE_105    ; Go if standard
        JMP     NON_STANDARD
;**********************|
;  select video mode   |
;**********************|
; Mode variables are standard. Select mode 105H using VESA BIOS
; subservice number 2
SET_MODE_105:
        MOV     BX,0105H        ; Mode number and high bit = 0
                                ; to request clear video
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,2            ; This subservice
        INT     10H             ; BIOS video service
; Test for valid returned value
        CMP     AX,004FH        ; Status for no error
        JE      OK_MODE_105     ; No error during mode set
        STC                     ; Error flag. Mode not set
        POP     DS              ; Restore caller's DS
        RET
OK_MODE_105:
        POP     DS              ; Restore caller's DS
        CLC                     ; No error flag
        RET
VESA_MODE_105   ENDP
P_CODE          ENDS
```

### 10.3.4 Subservice 3 — Get Video Mode

VESA BIOS subservice number 3 is used to obtain the current video mode. The VESA mode number is returned by the subservice in the BX register. The following code fragment shows a call to this VESA BIOS service:

```
;****************************************************************
; processing operations for obtaining the current VESA BIOS mode
;****************************************************************
P_CODE   SEGMENT PUBLIC
        ASSUME  CS:P_CODE
VESA_GET_MODE   PROC    FAR
; Procedure to obtain current VESA BIOS mode using subservice
; number 3
; VESA BIOS subservice number 3 to obtain current video mode
        MOV     AH,79           ; VESA BIOS service number
        MOV     AL,3            ; This subservice
        INT     10H             ; BIOS video service
; Test for valid returned value
        CMP     AX,004FH        ; Status for no error
        JE      MODE_OK         ; No error during mode set
;**********************|
;     ERROR exit       |
;**********************|
        STC                     ; Carry flag is set for error
        RET
MODE_OK:
        CLC                     ; No error
        RET
VESA_GET_MODE   ENDP
P_CODE          ENDS
```

## 10.3.5 Subservice 4 — Save/Restore Video State

VESA BIOS subservice number 4 is used to save and restore the state of the video system. This service, which is an extension of BIOS service number 28, is often used in a multitasking operating system to preserve the task states and by applications that manage two or more video environments. The subservice can be requested in three different modes, passed to the VESA BIOS routine in the DL register.

Mode number 0 (DL = 0) of subservice number 4 returns the size of the save/restore buffer. The four low bits of the CX register encode the machine state buffer to be reported. The bitmap for the various machine states is shown in Figure 10.6.

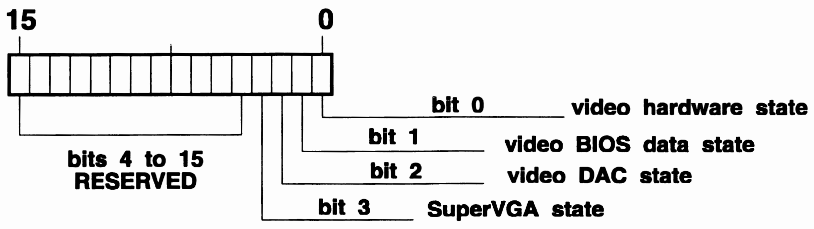


Figure 10.6 *VESA Machine State Bitmap*

The units of buffer size returned by mode number 0, of subservice number 4, are 64-byte blocks. The block count is found in the BX register.

Mode number 1 (DL = 1), of subservice number 4, saves the machine video state requested in the CX register (see Figure 8.6). The caller should provide a pointer to a buffer sufficiently large to hold the requested state data. The size of the buffer can be dynamically determined by means of a call using mode number 0, described above. The pointer to the buffer is passed in ES:BX.

Mode number 2 (DL = 2), of subservice number 4, restores the machine video state requested in the CX register (see Figure 8.6). The caller should provide a pointer to the buffer that holds data obtained by means of a call using mode number 1.

### 10.3.6 Subservice 5 — Switch Bank

VESA BIOS subservice number 5 is used to switch memory banks in those modes that require it. Software should call subservice number 1 to determine the size and address of the banks before calling this function. Two modes of this subservice are implemented: one to switch to a desired bank and another one to request the number of the currently selected bank.

Mode number 0 (BH = 0) is the switch bank command. The BL register is used by the caller to encode window A (value = 0) or window B (value = 1). The bank number is passed in the DX register. The following code fragment shows the necessary processing:

```
; VESA BIOS subservice number 5 register setup
        MOV     BX,0            ; Select bank in window A
                                ; and bank switch function
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     AX,4F05H        ; Service and subservice
        INT     10H
          .
          .
          .
```

Mode number 1 of subservice 5 (BH = 0) is used to obtain the number of the memory bank currently selected. The BL register is used by the caller to encode window A (value = 0) or window B (value = 1). The bank number is reported in the DX register.

Earlier in this section we mentioned that an application can also access the bank switching function in the BIOS by means of a far call to the service routine. The address of the service routine is placed in a far pointer variable by the successful execution of subservice number 1. For the far call operation the register setup for BH, BL, and DX is the same as for using subservice 5. However, in the far call version AH and AL need not be loaded, no meaningful information is returned, and AX and DX are destroyed.

### 10.3.7 Subservice 6 — Set/Get Logical Scan Line

VESA BIOS subservice number 6 is used to set or read the length of the logical scan line. Observe that the logical scan line can be wider than the physical scan line supported by the video hardware. This subservice was first implemented in VESA BIOS version 1.1. For this reason it is not available in the BIOS functions of earlier adapters.

### 10.3.8 Subservice 7 — Set/Get Display Start

VESA BIOS subservice number 7 is used to set or read from the logical page data the pixel to be displayed in the top-left screen corner. This subservice is useful to applications that use a logical screen that is larger than the physical display in order to facilitate panning and screen scrolling effects. As is the case with subservice number 6, this subservice was first implemented in VESA BIOS version 1.1. For this reason it is not available in the BIOS functions of many adapters.

### 10.3.9 Subservice 8 — Set/Get DAC Palette Control

VESA BIOS subservice number 8 was designed to facilitate programming of SuperVGA systems with more than 6-bit fields in the primary color registers of the DAC. The subservice contains two modes. Mode number 0 (BL = 0) is used to set a DAC color register width. The desired width value, in bits, is passed in the BH register by the caller. Mode number 1 (BL = 1) is used to obtain the current bit width for each primary color. The bit width is returned in the BH registers. The standard bit width for VGA systems is six.

This subservice was first implemented in version 1.2 of the VESA BIOS, released in October 22, 1991. Therefore it is not available in adapters with earlier versions of the VESA BIOS. Another feature introduced in VESA BIOS version 1.2 is the use of bit 0 of the CAPABILITIES field (see subservice 0 earlier in this section) to encode the presence of DAC registers capable of storing color encodings of more than six bits. Applications that propose to use subservice 8 should first test the low-order bit of the CAPABILITIES field to determine if this feature is implemented in the hardware.

## 10.4 SuperVGA Device Drivers

Direct hardware programming a particular SuperVGA chipset requires specific technical data from the manufacturer. The resulting code has limited portability to other systems. This approach is used in coding hardware-specific drivers that take full advantage of the capabilities of the system. An alternative method that ensures greater portability of the code, at a price in performance, is the use of the VESA BIOS services described starting at Section 10.3.

It is theoretically possible to design a general-purpose graphics routine that operates in every SuperVGA chipset and display mode. Here again, this universality can be achieved only at a substantial price in performance, an element that is usually critical to the animation programmer. For this reason the design and coding of mode-specific graphics routines is generally a more efficient approach. By using VESA BIOS functions it is possible to design mode-specific routines that are compatible with most SuperVGA systems that support the particular mode.

The procedures that follow use VESA BIOS mode number 105H with a resolution of 1024-by-768 pixels in 256 colors. We have selected this mode because it is compatible with modes used in the XGA system, and also because it is widely available in fully equipped SuperVGA adapters.

### 10.4.1 Address Calculations

Address calculations in a SuperVGA mode depend on the screen dimensions and the location of the video buffer in the system's memory space. In a mode-specific routine the number of pixels per row can be entered as a numeric value. In modes that require more than one memory bank the bank size must also enter into the address calculations. Most SuperVGA adapters use a bank size of 64K, which can be hard-coded in the address calculation routine. On the other hand, it is possible to use a memory variable that stores the number of pixels per row and the bank size parameters in order to design address calculation routines that work in more than one mode. In the following code fragment we have assumed that the SuperVGA is in VESA mode 105H, with 1024 pixels per scan line, and that the bank size is 64K. The display routines assume that the base address of the video buffer is A000H.

```
; Calculate pixel address from the following coordinates:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
;
; Code assumes:
;       1. SVGA is in a 1024-by-768 pixel mode in 256 colors
;          (mode number 105H)
;       2. Bank size is 64K
;
; Get address in SVGA memory space
        CLC                         ; Clear carry flag
        PUSH    AX                  ; Save color value
        MOV     AX,1024             ; Pixels per scan line
        MUL     DX                  ; DX holds line count of address
        ADD     AX,CX               ; Add pixels in current line
        ADC     DX,0                ; Answer in DX:AX
                                    ; DL = bank, AX = offset
        MOV     BX,AX               ; Offset to BX
```

At this point BX holds the pixel offset and DX the bank number. Note that the pixel offset is the offset within the selected bank, and not the offset from the start of the screen as is often the case in VGA routines.

### 10.4.2 Bank Switching Operations

In a SuperVGA adapter set to VESA mode number 105H (resolution of 1024-by-768 pixels in 256 colors) the number of video memory banks depends on the bank size. With a typical bank size of 64K the entire video memory space requires 12 memory banks, since:

$$\frac{1024 * 768}{65536} = 12$$

In order to update the entire video screen the software has to perform 12 bank switches. This would be the case in performing a clear screen operation. Furthermore, many relatively small screen objects cross one or more bank boundaries. In fact, in VESA SuperVGA mode 105H any graphics object or window that exceeds 64 pixels in height necessarily overflows one bank.

For these reasons bank switching operations should be optimized to perform their function as quickly as possible. The ideal solution would be to embed the hardware bank switching code within the address calculation routine. This method is similar to the one described for the XGA (see Chapter 9). However, XGA software does not have to contend with variations in hardware. We have seen that in the SuperVGA environment to hard-code the bank switching operation would almost certainly make the routine not portable to other devices. An alternative solution is to perform bank switching by means of VESA BIOS service number 5, described in Section 10.3.6. The following code fragment shows the code for bank switching using the VESA BIOS service:

```
;*********************|
;    change banks     |
*********************|
; Select video bank using VESA BIOS subservice number 5
; VESA BIOS subservice number 5 register setup
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0            ; Select bank in window A
        MOV     AX,4F05H        ; Service and subservice
        INT     10H
```

An alternative option that would improve performance of the bank switching operation is by means of a far call to the service routine, as mentioned in Section 10.3.1. The following code fragment shows bank switching using the far call method. The code assumes that the address of the service routine is stored in a doubleword variable named BANK_FUN. This address can be obtained by means of VESA BIOS subservice number 1 (get mode information) discussed in Section 10.3.2

```
.
;********************|
;    change banks     |
;  by far call method |
;********************|
; Select video bank by means of a far call to the bank switching
; routine provided by the chipset manufacturer
; Code assumes that the far address of the service routine is
; stored in a doubleword variable named BANK_FUN
; Register setup for far call method
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0            ; Select bank in window A
        PUSH    AX              ; Preserve caller's context
        PUSH    DX
        CALL    DWORD PTR BANK_FUN
        POP     DX              ; Restore context
        POP     AX
```

Observe that to use the far call method the doubleword variable that holds the address of the service routine must be reachable at the time of the call. Therefore, if the variable is in another segment, a segment override byte is required.

## 10.5 SuperVGA Pixel Level Operations

Once the pixel address has been determined and the hardware has been switched to the corresponding video memory bank, setting the pixel is a simple write operation. For example, in VESA mode number 105H, once the address calculation and the bank switching routine listed in Section 10.4.2 have executed, the pixel can be set by means of the instruction

```
        MOV     BYTE PTR ES:[BX],AL
```

The code assumes that ES holds the base address of the video buffer, BX the offset within the bank, and AL the 8-bit color code. Note that since VESA mode number 105H is not a planar mode, no previous read operation is necessary to enable the latching mechanism.

Reading a pixel in a SuperVGA mode is usually based on the same address and bank switching operations as those required for setting a pixel. The actual read instruction is in the form

```
        MOV     AL,BYTE PTR ES:[BX]
```

### 10.5.1 SuperVGA Pixel Write

The following procedure performs a pixel write operation while in SuperVGA mode number 105H:

```
;****************************************************************
;    processing operations for pixel setting using VESA BIOS
;****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
SVGA_PIX_105   PROC  FAR
; Write a screen pixel accessing SVGA memory directly and using
; VESA BIOS service to select bank
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
;       BL = pixel color in 8-bit format
; Code assumes:
;       1. SVGA is in a 1024-by-768 pixel mode in 256 colors
;           (mode number 105H)
;       2. video bank size is 64K
;       3. ES holds base address of video buffer (A000H)
;
;**********************|
;   save caller's DS   |
; set DS to SVGA_DATA  |
;**********************|
; This manipulation is required in order access the address of
; the bank switching routine
        PUSH    DS
        MOV     AX,SVGA_DATA    ; Local data segment
        MOV     DS,AX           ; to DS
        ASSUME  DS:SVGA_DATA
;
        PUSH    BX              ; Save entry registers
        PUSH    CX
        PUSH    DX
        MOV     AL,BL           ; Color to AL
; Get address in SVGA memory space
        CLC                     ; Clear carry flag
        PUSH    AX              ; Save color value
        MOV     AX,1024         ; Pixels per scan line
        MUL     DX              ; DX holds line count of address
        ADD     AX,CX           ; Add pixels in current line
        ADC     DX,0            ; Answer in DX:AX
                                ; DL = bank, AX = offset
        MOV     BX,AX           ; Offset to BX
;**********************|
;    change banks      |
;**********************|
; Bank switching is performed by means of a far call to the
; service routine located at the label SWITCH_BANK
        PUSH    BX              ; Save entry registers
        PUSH    DX
; VESA BIOS subservice number 5 register setup:
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0            ; Select bank in window A
        CALL    DWORD PTR SWITCH_BANK
        POP     DX              ; Restore entry registers
        POP     BX
        POP     AX              ; Restore color
;*********************|
;   set the pixel     |
;*********************|
        MOV     ES:[BX],AL      ; Write the dot
        POP     DX              ; Restore entry values
        POP     CX
        POP     BX
        POP     DS              ; Restore caller's DS
        RET
SVGA_PIX_105    ENDP
P_CODE          ENDS
```

### 10.5.2 SuperVGA Pixel Read

The following procedure can be used to read a screen pixel in this same mode:

```
;****************************************************************
;   processing operations for reading a pixel using VESA BIOS
;****************************************************************
;
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
SVGA_READ_105   PROC    FAR
; Read a screen pixel accessing SVGA memory directly and using
; VESA BIOS service to select bank
; On entry:
;       CX = x coordinate of pixel
;       DX = y coordinate of pixel
; Code assumes:
;       1. SVGA is in a 1024-by-768 pixel mode in 256 colors
;          (mode number 105H)
;       2. video bank size is 64K
;       3. ES holds base address of video buffer (A000H)
; On exit:
;       AL = 8-bit pixel color code
;
```

```
;*********************|
;   save caller's DS   |
; set DS to SVGA_DATA  |
;*********************|
; This manipulation is required in order access the address of
; the bank switching routine
        PUSH    DS
        MOV     AX,SVGA_DATA     ; Local data segment
        MOV     DS,AX            ; to DS
        ASSUME  DS:SVGA_DATA
;
        PUSH    BX               ; Save entry registers
        PUSH    CX
        PUSH    DX
; Get address in SVGA memory space
        CLC                      ; Clear carry flag
        MOV     AX,1024          ; Pixels per scan line
        MUL     DX               ; DX holds line count of address
        ADD     AX,CX            ; Add pixels in current line
        ADC     DX,0             ; Answer in DX:AX
                                 ; DL = bank, AX = offset
        MOV     BX,AX            ; Offset to BX
;
;*********************|
;    change banks      |
;*********************|
; Bank switching is performed by means of a far call to the
; service routine located at the label SWITCH_BANK
        PUSH    BX               ; Save entry registers
        PUSH    DX
; VESA BIOS subservice number 5 register setup:
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0             ; Select bank in window A
        CALL    DWORD PTR SWITCH_BANK
        POP     DX               ; Restore entry registers
        POP     BX
;
;*********************|
;   read the pixel     |
;*********************|
        MOV     AL,BYTE PTR ES:[BX]    ; Read memory
        POP     DX               ; Restore entry values
        POP     CX
        POP     BX
        POP     DS               ; Restore caller's DS
        RET
SVGA_READ_105   ENDP
P_CODE  ENDS
```

## 10.5.3 Clearing the SuperVGA Screen

Clearing the video display is another primitive function frequently useful to a graphics application. The following procedure can be used to clear the screen in SuperVGA mode 105H:

```
;*****************************************************************
; processing operations for clearing the screen using VESA BIOS
;*****************************************************************
P_CODE  SEGMENT PUBLIC
        ASSUME  CS:P_CODE
;
SVGA_CLS_105     PROC    FAR
; Clear video memory while in SVGA mode number 105H
; On entry:
;         AL = 8-bit color code to use for initialization
; Code assumes:
;       1. SVGA is in a 1024-by-768 pixel mode in 256 colors
;           (mode number 105H)
;       2. Video bank size is 64K
;       3. ES holds base address of video buffer (A000H)
;
;**********************|
;   save caller's DS   |
; set DS to SVGA_DATA  |
;**********************|
; This manipulation is required in order to access the address of
; the bank switching routine
        PUSH    DS
        MOV     DX,SVGA_DATA     ; Local data segment
        MOV     DS,DX            ; to DS
        ASSUME  DS:SVGA_DATA
; BL holds bank number
        MOV     DX,0             ; Initialize to first bank
;**********************|
;    change banks      |
;**********************|
; Bank switching is performed by means of a far call to the
; service routine located at the label SWITCH_BANK
NEXT_BANK:
        PUSH    AX               ; Save color code
        PUSH    BX               ; Save entry registers
        PUSH    DX
; VESA BIOS subservice number 5 register setup:
; BH = 0 to select bank
; BL = 0 to select window A
; DX = bank number
        MOV     BX,0             ; Select bank in window A
        CALL    DWORD PTR SWITCH_BANK
        POP     DX               ; Restore entry registers
        POP     BX
        POP     AX                ; Restore color code
; Write 65536 bytes of 00H in current bank
        MOV     CX,0FFFFH         ; CX is byte counter
        CLD                       ; Forward direction
        MOV     DI,0              ; Start of block
        REP     STOSB             ; Store 65536 bytes
; Bump bank
        INC     DX                ; Next bank
        CMP     DX,12             ; 12 is last bank
        JNE     NEXT_BANK
        POP     DS                ; Restore caller's DS
        RET
SVGA_CLS_105    ENDP
P_CODE  ENDS
```

Part

3

# Animation Techniques

Chapter

11

# Background, Objects, and Text

## 11.0 Background Techniques

An animated application such as an electronic video game, a trip to a foreign galaxy, or a cartoon-like short subject usually requires a background on which the animation takes place. In addition, this background often also partakes in the animated action. For example, in a simulated space trip, the celestial objects that constitute the background can be animated by translation (panning and zooming) as the spacecraft moves to its imaginary destination.

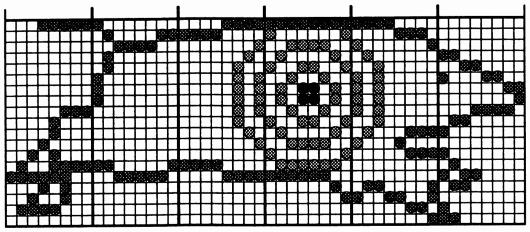
Regarding backgrounds the animation programmer has two main concerns: acquiring the background image or images, and manipulating these images.

## 11.1 Bitmap Backgrounds

Like all graphics images backgrounds can be in vector form or encoded as a bitmap. In Section 4.3 we discussed the problems and methodology related to the acquisition of bit-mapped images. It is safe to state that most backgrounds for animated applications are in the form of bitmaps. Therefore the discussion in Section 4.3 can be related to acquiring bitmaps to be used as background. The most used methods of bitmap acquisition are described in the following sections.

### 11.1.1 Hand Bit-Coding a Bitmap Background

One straightforward method of generating the background image is for the programmer to directly create it in the application's memory space by defining every pixel in the image. The process is similar to that of creating a memory resident bitmap by bit coding. The process consists of drawing the image as individual dots, and then calculating the attribute codes that represent each dot. The method is shown in Figure 4.2.



Pixel attributes:
◎ = blue
◉ = green
● = red

Figure 11.1 *Pixel Drawing for Bitmap Generation*

For large images, hand bit-coding can be a laborious and time-consuming process. Therefore it is more often used for generating small objects and local backgrounds than for extended screen areas. In any case, it offers the advantage that the programmer directly controls the placement and attribute of each screen pixel and can easily make modifications in the image.

In generating a hand-coded bitmap it is convenient to start with a color drawing on quadrille paper. From this drawing, the programmer can hand-code the corresponding bitmap. Figure 11.1 shows a color-coded drawing of a running boar target from which the memory-resident bitmap can be derived.

### 11.1.2 Electronic Drawing Methods

A more convenient alternative than hand bit-coding the background image is using a draw or paint program to create the image and then storing it as a bitmap in a standard format that can be read by the program. Many commercial and shareware draw and paint applications are available on the market. The one requirement is that the software be capable of storing the image in a bitmap format compatible with the program. Electronic drawing is perhaps the most used method of generating backgrounds.

### 11.1.3 Ray-Tracing a Bitmap Background

A mathematical rendering technique known as ray tracing makes possible the generation of realistic background images in three dimensions. The images are particularly suited for animated electronic games in which the player moves through mazes, castles, or hallways, as in many popular commercial and shareware programs. The image resulting from the ray-traced application is also in the form of a bitmap encoded in a standard format (see Chapter 4).

One of the most popular ray-tracing applications is a program named *POV-Ray* (Persistence of Vision Ray Tracer), of which a shareware version is available. The POV-Ray program files are found on the Graphics Developer Forum of Compuserve (GO GRAPHDEV), on the America On-Line service (keyword PCGRAPHICS), and on many *bulletin board systems* (BBS). Figure

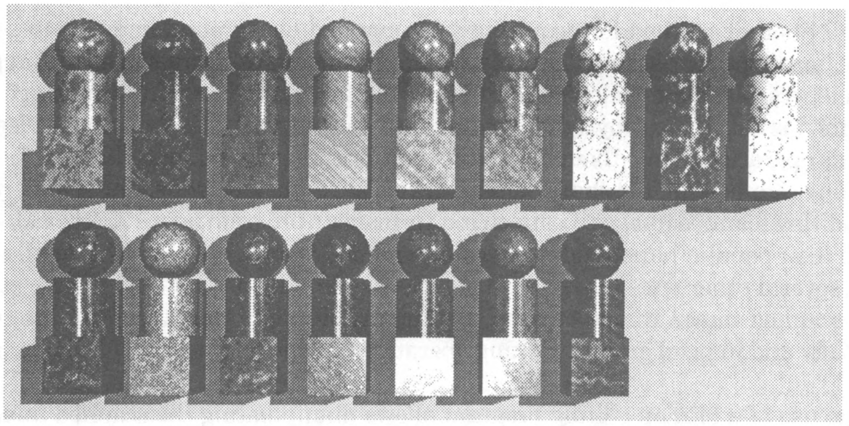


Figure 11.2 *Sample Texture File from the POV-Ray Program*

11.2 is a print of one of the sample files furnished with the POV-Ray program. Please notice that the original file is in color; therefore much of the original detail is lost in the monochrome reproduction.

Learning the use of a ray-tracing application is much like learning a programming language. The seed image is usually defined in parametric form. The user also enters the operational mode which is applied during the ray-trace operation. The ray-trace application then goes to work in creating a derivative image. Once generated, the image can be stored in a standard bitmap format selected by the user. Figure 11.3 is a ray-traced image used in a maze that is part of an electronic game program. The program is being developed by two students at Montana State University, Northern: Dale Niemeyer and David Oard.

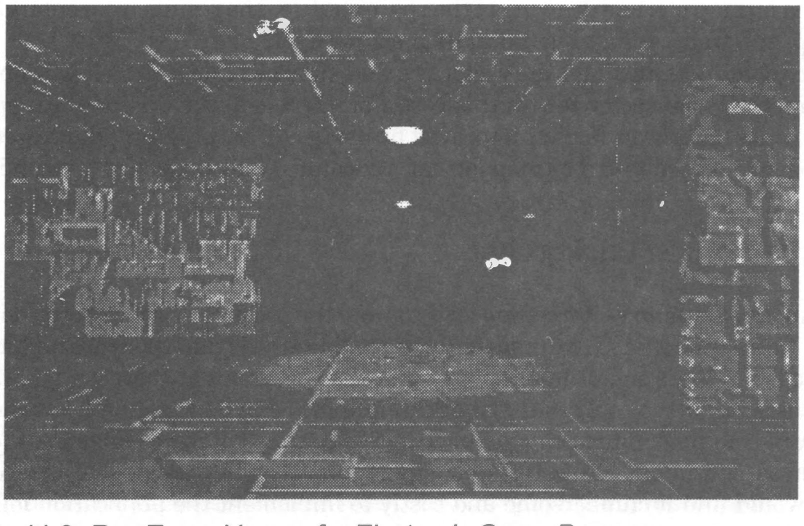


Figure 11.3 *Ray-Traced Image for Electronic Game Program*

### 11.1.4 Scanning a Bitmap Background

Still another method for acquiring a bit-mapped background image is to scan it from hard copy. The scanned print can be an original art work created by the program developers, a commercial print, or a public domain reproduction. If the image to be scanned is subject to copyright, it would be a good idea to first check on the legality of this operation. Some thoughts on this matter can be found in Section 4.3.1.

Many commercial and shareware programs are available for driving scanners and for editing scanned images. Logitech Inc. furnishes a program named *Fototouch Color* with their popular hand-held color scanner named *Scanman Color*. This application, like several other ones in popular use, allows scanning a hard copy image, storing the scanned image in one of several standard bitmap formats, editing the image globally and in pixel groups. Editing operations can go to the level of a single image pixel.

The use of a bitmap editing program allows manipulating the scanned image by cropping and by altering the acquired bitmap. These manipulations are often necessary since rarely can a scanned image be used as originally acquired. Here again, the reader should consider the pertinent legal issues before using proprietary or copyrighted images.

### 11.1.5 Multiple Image Manipulations

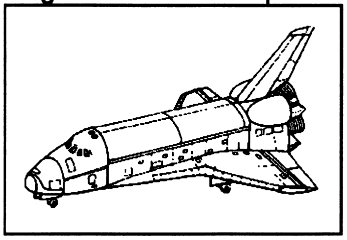
Some of the image acquisition and editing methods described in Section 11.1.4 can be consecutively applied in order to generate a specific effect. We have already mentioned that scanned images are usually modified by means of bitmap editing programs. Another type of application, sometimes called bitmap tracer or vectorizer, creates a vector image from a bitmap. One advantage of the vector image is that it can be rotated and scaled without distortion or loss of detail. A scaled bitmap, on the other hand, is usually grainy and visually unpleasant. Figure 11.4 shows the original scanned bitmap of a line drawing of the space shuttle, the results from enlarging the bitmap, and from vectorizing it and then enlarging it.

Notice that in Figure 11.4 the vectorized and enlarged image is as it was obtained from applying the vectorizer (in this case CorelTrace) to the original bitmap. This image could have been further improved by using a draw program; the result could have been exported into a bitmap format, or scanned and retouched a second time.

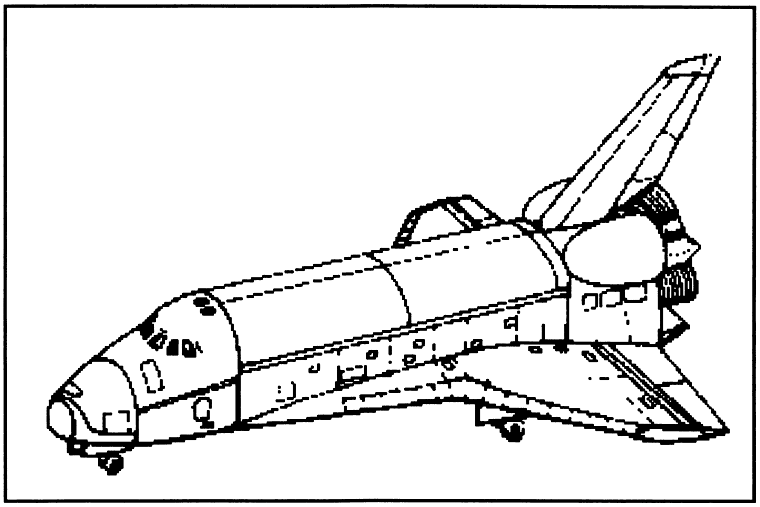
## 11.2 Vectorized Background

The subject of bitmap vectorization leads us into the possible use of vectorized images in graphics applications. The main difficulty in this area is that there are no generally accepted standards and utilities for vector image operations as there are for bitmaps, notwithstanding that substantial work in graphics standardization has been done in Europe and the United States. But protocols such as GKS are elaborate and their use is more suitable at the system and language levels. The developer of an electronic game would find it cumbersome and costly to implement the application following the GKS protocol, particularly when considering that the few graphics manipulations

Original scanned bitmap



Enlarged bitmap



Vectorized and enlarged image

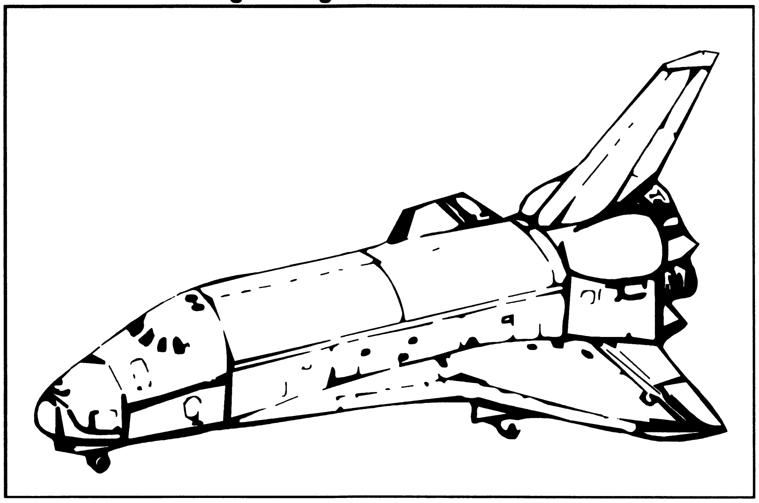


Figure 11.4 *Multiple Manipulations*

necessary to an application of this type can usually be furnished more efficiently, and with less programming effort, by customizing the code.

Therefore, in developing an animated application that requires vector image operations the program designer first has to decide whether to adopt an existing graphics standard or to use a customized approach. Here again, the factors of performance, portability, and development cost usually determine the final decision.

### 11.2.1 Vector/Bitmap Background

A vectorized representation of a background is often not possible. In most PC video systems it is possible to store a bitmap image that exceeds the dimensions of the viewport. This technique allows the use of very satisfactory panning operations by changing the address of the video buffer that is mapped to the screen. In Chapter 7 we presented vertical and horizontal panning operations for VGA mode X. These routines can be readily adapted to other VGA modes and to XGA systems.

Therefore, the PC animator often finds effective techniques for panning a bitmap background. However, bitmaps cannot be scaled, rotated, or mathematically transformed with the same ease as a vector image. If the program can be designed to avoid other background transformations, except panning, then the bitmap background works satisfactorily. On the other hand, if the application requires scaling (zooming), rotation, or other geometrical transformations of the background image, then a vector-based background is necessary. The reader can find vector-based graphics primitives in our books *Graphics Programming Solutions* and *High Resolution Video Graphics* listed in the Bibliography.

Sometimes an application is able to combine bitmap and vector techniques in a single image. This situation often occurs in background images that contain several, rather small, bitmaps. For example, a graphics application that displays stars on a night sky can deal with a few dozen objects over a uniform and extended background. In this case, the program designers can define several bitmaps to represent stars of different magnitudes. Since the individual stars are positioned on the sky using a coordinate system, then each individual star can be described in the database in terms of its coordinates and magnitude. The magnitude, in turn, determines which of the available bitmaps is selected at display time. The result is a vector/bitmap approach in which the object's position is defined as a vector and its image is defined as a bitmap.

#### Multiple Bitmaps

In the case of the representation of the night sky, mentioned in the preceding paragraph, the program designer can create several star image bitmaps according to the object's magnitude. For example, if the application is to handle stars of eight different magnitudes (conventionally, magnitude 0 is the brightest), then the bitmaps can be as shown in Figure 11.5.

Notice that all eight bitmaps in Figure 11.5 have a common center, located five pixels to the right and four pixels down from the top-left corner. Giving all

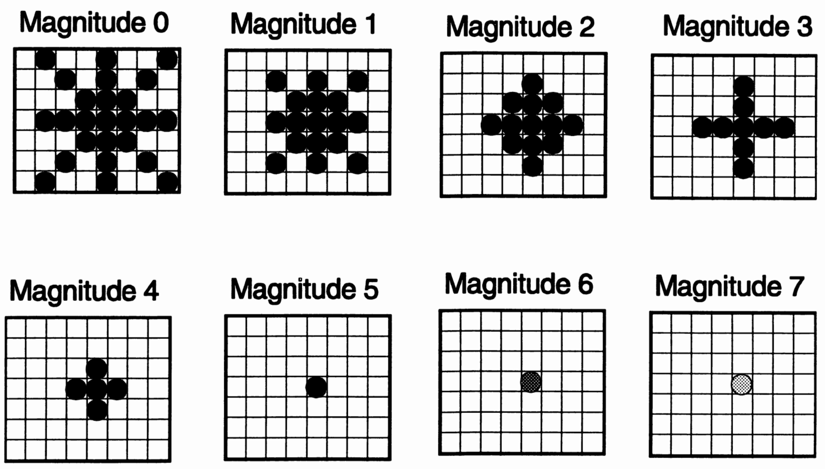


Figure 11.5 *Bitmaps for Eight Star Magnitudes*

images a common geometrical center simplifies the calculations at display time since the same constant is subtracted from the pixel coordinates in every case.

## Creating the Star Database

The data encoding a vector/bitmap object must express two parameters: the object's vector address, and a means for locating the object's bitmap. In the case of these star objects, the vector element is expressed in terms of the star's declination and right ascension, which is the conventional coordinate system used in astronomy (equatorial coordinates). We sometimes use the abbreviation DEC to represent declination and RA to represent right ascension. Without getting into astronomical details, we should mention that declination is the object's north-south coordinate, often equated with the object's latitude. The declination is usually expressed in degrees. The right ascension is the object's position in relation to the vernal equinox (midnight, March 21). The right ascension is usually expressed in hours and minutes.

The information regarding each object included in the database depends on the purpose of the application. For example, an astronomy program may require the object's name or identification, the name of the constellation to which the object belongs, the object type (that is, if the object is a star, star cluster, nebula, galaxy, or other), its visual color, and its distance from earth. In this case one possible approach is to organize the objects by the constellations to which they belong.

This scheme would make a constellation the unit of database storage. Therefore the database must also encode information regarding the constellation, such as the constellation name or number, the number of objects in it, and the attribute or color with which it is displayed. The following code fragment shows the memory storage of data representing seven stars in the constellation Ursa Minor (Little Dipper):

```
;****************************************************************
;                    constellation data
;****************************************************************
;
; Constellation data format
;       OFFSET      UNIT      CONTENTS
;         0         byte      Constellation number
;         1         byte      Number of objects in Constellation
;         2         byte      Default display color
;         3         byte      RESERVED
;
;****************************************************************
;              celestial objects in constellation
;****************************************************************
;
; Celestial object data format:
;       OFFSET      UNIT      CONTENTS
;         0         byte      Star number in constellation
;                             (1-based)
;         1         byte      Object type:
;                                1 = star
;                                2 = star cluster
;                                3 = nebula
;                                4 = galaxy
;       2-17       string     Star name (ASCII)
;        18         byte      Integer of magnitude (binary)
;        19         byte      Fraction of magnitude (binary)
;        20         word      RA hours (binary)
;        22         word      RA minutes (binary)
;        24         word      DEC degrees (binary)
;        26         word      DEC minutes (binary)
;       28-33      string     Star code (ASCII)
;        34         word      Storage for x screen coordinate
;        36         word      Storage for y screen coordinate
;       +38   -------------- start of next object
;
URSA_MINOR      DB      1       ; Constellation number
                DB      7       ; Number of objects
                DB      00001001B       ; Constellation color
                                        ; is bright blue
                DB      0       ; Reserved
; Start of first star (in this case Polaris)
                DB      1       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Polaris         ' ; 16-character name
                                ; plus / terminator
                DB      1       ; Magnitude
                DB      9       ; Fraction of magnitude
                DW      2       ; RA hours
                DW      31      ; RA minutes
                DW      89      ; DEC degrees (signed)
                DW      15      ; DEC minutes
                DB      'S18   '; Star code (6 digits)
                DW      0       ; Storage for x coordinate
                DW      0       ; Storage for y coordinate
;
; Next star
                DB      2       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Kochab          ' ; 16-character name
                                ; plus / terminator
                DB      2       ; Magnitude
                DB      1
                DW      14      ; RA hours
                DW      50      ; RA minutes
                DW      74      ; DEC degrees
                DW      10      ; DEC minutes
                DB      'S156  '; Star code (6 digits)
                DW      0       ; Coordinates
                DW      0
;
; Next star
                DB      3       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Pherkad         ' ; 16-character name
                                ; plus / terminator
                DB      3       ; Magnitude
                DB      1
                DW      15      ; RA hours
                DW      20      ; RA minutes
                DW      71      ; DEC degrees
                DW      50      ; DEC minutes
                DB      'S163  '; Star code (6 digits)
                DW      0
                DW      0
;
; Next star
                DB      4       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Yildun          ' ; 16-character name
                                ; plus / terminator
                DB      4       ; Magnitude
                DB      0
                DW      17      ; RA hours
                DW      50      ; RA minutes
                DW      86      ; DEC degrees
                DW      0       ; DEC minutes
                DB      'None  '; Star code (6 digits)
                DW      0
                DW      0
; Next star
                DB      5       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Epsilon UMI     ' ; 16-character name
                                ; plus / terminator
                DB      4       ; Magnitude
                DB      0
                DW      17      ; RA hours
                DW      0       ; RA minutes
                DW      +82     ; DEC degrees
                DW      0       ; DEC minutes
                DB      'None  '; Star code (6 digits)
                DW      0
                DW      0
; Next star
                DB      6       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Zeta UMI        ' ; 16-character name
                                ; plus / terminator
                DB      4       ; Magnitude
                DB      0
                DW      15      ; RA hours
                DW      50      ; RA minutes
                DW      +78     ; DEC degrees
                DW      0       ; DEC minutes
                DB      'None  '; Star code (6 digits)
                DW      0
                DW      0
; Next star
                DB      7       ; Star number in constellation
                                ; (1-based)
                DB      1       ; Object type code
                DB      'Eta UMI         ' ; 16-character name
                                ; plus / terminator
                DB      4       ; Magnitude
                DB      0
                DW      16      ; RA hours
                DW      20      ; RA minutes
                DW      +76     ; DEC degrees
                DW      0       ; DEC minutes
                DB      'None  '; Star code (6 digits)
                DW      0
                DW      0
;
                DB      00FEH   ; End of constellation mark
; End of data
```

Notice that the constellation name is not stored in the database. The code can obtain the name by using the constellation number as an index into a list of constellation names. The following list holds the names of eight constellations, starting with Ursa Minor:

```
; List of Constellation names (20 characters per entry)
CONST_NAMES     DB      'URSA MINOR         '
                DB      'URSA MAJOR         '
                DB      'ORION              '
                DB      'BOOTES             '
                DB      'CASSIOPEIA         '
                DB      'LYRE               '
                DB      'GEMINI             '
                DB      'HERCULES           '
```

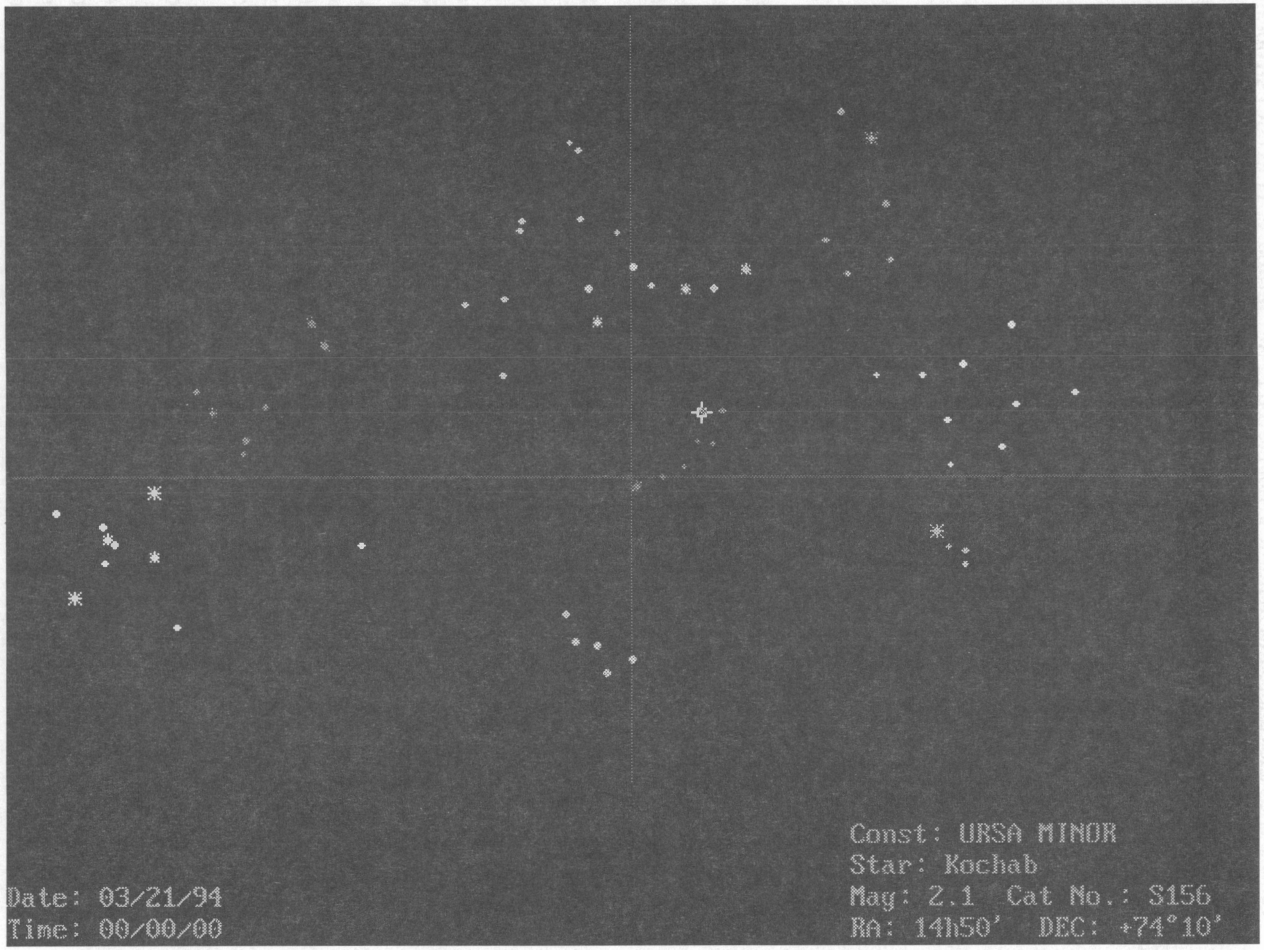
## Display of Vector/Bitmap Object

Code can display vector/bitmap objects by obtaining from the database the object's parameter that is used in selecting the corresponding bitmap. In the case of the listed star database, it is the object's magnitude that determines the bitmap to use. This item is located at offset 3 in the object's data field. The display location is determined by the object's vector data. In the case of the star database example this data would be the object's declination and right ascension. These data items are located starting at offset 5 of the object's data field.

Figure 11.6 is a screen dump of a demonstration program that displays the principal objects in eight circumpolar constellations. The original program, named *Astrium*, is furnished with this book's diskette option. In the original, the constellations are color-coded, but they are difficult to distinguish in the monochrome print. The data displayed on the left-bottom part of the screen is the data and time information that is necessary to locate the constellation on the celestial sphere. The data displayed at the right-bottom part of the screen corresponds to the object that is closest to the position of the cursor at the time that the left mouse button is pressed. In Figure 11.6 the cursor is closest to the star named Kochab in Ursa Minor. The constellation name, magnitude, catalog number, right ascension (RA), and declination (DEC) are obtained from the database.

Since the objects manipulated by the Astrium demonstration program are defined in vector/bitmap form, it is possible to scale, rotate, and translate them. Therefore if the user wishes to see the position of the stars at a different date and time, the software can rotate the constellations accordingly by performing a mathematical operation on the coordinate points. Notice that if the constellations had been defined as a single bitmap, rotation, translation, and scaling would be much more complicated and time-consuming operations.

The date and time data entered by the user into the Astrium program is in the form of two ASCII strings. One holds the current date in mm/dd/yy format, and the second one holds the hour in hh/mm/ss format. Before this data can be used to rotate the coordinates of the constellations to be displayed, it must first be converted into



```
                                        Const: URSA MINOR
                                        Star: Kochab
Date: 03/21/94                          Mag: 2.1   Cat No.: S156
Time: 00/00/00                          RA: 14h50'   DEC: +74°10'
```

Figure 11.6 *Initial Screen of the Astrium Program*

**fractional degrees and stored in the coprocessor's ST(0) register. The following procedures perform the necessary operations:**

```
;****************************************************************
;       procedures to convert ASCII positional data into
;                  decimal form in ST(0)
;****************************************************************
;
DATA    SEGMENT BYTE    PUBLIC
ASCII_BUF       DB      '     ',0H      ; Buffer for ASCII digits
;
; Binary data
BIN_MONTH       DW      0       ; Binary month number in date
BIN_DAYS        DW      0       ; Binary day number in date
THIRTY          DQ      30.0    ; Month to degree conversion
BIN_HOURS       DW      0       ; Hours in binary
BIN_MINUTES     DW      0       ; Minutes in binary
EQNX_DAYS       DW      0       ; Days from spring equinox
DEGS_PER_DAY    DT      0.98630137  ; Degrees per solar day
DEGS_PER_HOUR   DQ      15.0    ; Degrees in one hour
DEGS_PER_MIN    DQ      0.25    ; Degrees in one minute
;
DAYS_TABLE      DW      0       ; January - 31
                DW      31      ; February - 28 (not leap year)
                DW      59      ; March - 31
                DW      90      ; April - 30
                DW      120     ; May - 31
                DW      151     ; June - 30
                DW      181     ; July - 31
                DW      212     ; August - 31
                DW      243     ; September - 30
                DW      273     ; October - 31
                DW      304     ; November - 30
                DW      334     ; December - 31
DATA    ENDS
;
;****************************************************************
;                          code segment
;****************************************************************
;
CODE    SEGMENT BYTE    PUBLIC
        ASSUME  CS:CODE, DS:DATA
;****************************************************************
;                          procedures
;****************************************************************
;
TIME_TO_DEG     PROC    NEAR
; Convert time in hh/mm/ss format to degrees from hour 0.0
;
; On entry:
```

```
;         DS:SI -- Buffer holding hh/mm/ss
; On exit:
;         ST(0) = fractional degrees, counterclockwise from
;                 0.0h
;
; At this point DS:SI -- buffer holding string
; Get month digits into ASCII_BUF
        LEA     DI,ASCII_BUF ; Pointer to destination
MOVE_HOURS:
        MOV     AL,[SI]         ; Get digit
        CMP     AL,'/'          ; Test for end of field
        JE      HOURS_MOVED     ; Go if at end
        MOV     [DI],AL         ; Digit to buffer
        INC     DI              ; Bump pointers
        INC     SI
        JMP     MOVE_HOURS      ; Continue
HOURS_MOVED:
        MOV     BYTE PTR [DI],20H       ; Space at end of buffer
        LEA     BX,ASCII_BUF    ; Setup pointer
        CALL    ASC_TO_BIN      ; Library routine to convert
                                ; ASCII to binary
; DX has binary hours
        MOV     BIN_HOURS,DX    ; Store month number
; Move minutes field into buffer
        INC     SI              ; Bump pointer to days field
        LEA     DI,ASCII_BUF    ; Pointer to destination
MOVE_MINS:
        MOV     AL,[SI]         ; Get digit
        CMP     AL,'/'          ; Test for end of field
        JE      MINS_MOVED      ; Go if at end
        MOV     [DI],AL         ; Digit to buffer
        INC     DI              ; Bump pointers
        INC     SI
        JMP     MOVE_MINS       ; Continue
MINS_MOVED:
        MOV     BYTE PTR [DI],20H       ; Space at end of buffer
        LEA     BX,ASCII_BUF    ; Setup pointer
        CALL    ASC_TO_BIN      ; Library routine
; DX has binary minutes
        MOV     BIN_MINUTES,DX  ; Store minutes
; At this point:
;    BIN_HOURS = binary for hours number
;    BIN_MINUTES = binary for minutes number
; 1 hour = 15 degrees
; 1 minute = 0.25 degrees
;                               |   ST(0)   |  ST(1)   |  ST(2)   |
        FLD     DEGS_PER_HOUR   ;   15      | -------  |
        FILD    BIN_HOURS       ;    h      |   15     | -------  |
        FMULP   ST(1),ST        ; h * 15    | -------  |
        FLD     DEGS_PER_MIN    ;  0.25     |  h * 15  | -------  |
        FILD    BIN_MINUTES     ;   m       |   0.25   |  h * 15  |
        FMULP   ST(1),ST        ; m * 0.25 |  h * 15 | ------- |
        FADD                    ;  degs    | ------- |
        RET
TIME_TO_DEG     ENDP
;
DATE_TO_DEG     PROC    NEAR
; Convert date in mm/dd format to days from spring equinox
; and to fractional degrees
; Spring equinox at 0.0h March 21
;
; On entry:
;         DS:SI -- Buffer holding mm/dd/ value
; On exit:
;         ST(0) = fractional degrees, counterclockwise from
;                 0.0h spring equinox
;
; At this point DS:SI -- buffer holding string
;
; Get month digits into ASCII_BUF
        LEA     DI,ASCII_BUF ; Pointer to destination
MOVE_MONTH:
        MOV     AL,[SI]         ; Get digit
        CMP     AL,'/'          ; Test for end of field
        JE      MONTH_MOVED     ; Go if at end
        MOV     [DI],AL         ; Digit to buffer
        INC     DI              ; Bump pointers
        INC     SI
        JMP     MOVE_MONTH      ; Continue
MONTH_MOVED:
        MOV     BYTE PTR [DI],20H       ; Space at end of buffer
        LEA     BX,ASCII_BUF    ; Setup pointer
        CALL    ASC_TO_BIN      ; Library routine
; DX has binary month number
        MOV     BIN_MONTH,DX    ; Store month number
; Move days field into buffer
        INC     SI              ; Bump pointer to days field
        LEA     DI,ASCII_BUF    ; Pointer to destination
MOVE_DAYS:
        MOV     AL,[SI]         ; Get digit
        CMP     AL,'/'          ; Test for end of field
        JE      DAYS_MOVED      ; Go if at end
        MOV     [DI],AL         ; Digit to buffer
        INC     DI              ; Bump pointers
        INC     SI
        JMP     MOVE_DAYS       ; Continue
DAYS_MOVED:
        MOV     BYTE PTR [DI],20H       ; Space at end of buffer
        LEA     BX,ASCII_BUF    ; Setup pointer
        CALL    ASC_TO_BIN      ; Library routine
; DX has binary day number
        MOV     BIN_DAYS,DX     ; Store day number
```

```
; At this point:
;    BIN_MONTH = binary month number
;    BIN_DAYS = binary days number
; DAYS_TABLE lists the number of days at the start of each month
        MOV     AX,BIN_MONTH     ; Binary month to AX
        LEA     SI,DAYS_TABLE    ; Listing of days
        DEC     AX               ; Reduce to range
        ADD     AX,AX            ; Double to get word offset
        ADD     SI,AX            ; Add offset to pointer
        MOV     BX,[SI]          ; BX holds number of days
                                 ; until the present month
        MOV     AX,BIN_DAYS      ; Day number to AX
        ADD     AX,BX            ; Add days
; AX holds days from January 1, 0.0 hours, to present day
; Spring equinox is 80 days from this date
        CMP     AX,80            ; Test for equinox date
        JAE     PAST_EQUINOX     ; Go if past equinox
; At this point the current date precedes the equinox
; There are 285 days from equinox to January 1
; Therefore days past equinox are 285 + AX
        MOV     BX,285           ; Equinox to year end
        ADD     AX,BX            ; AX has days from equinox
        JMP     DAYS_DONE        ; Go to exit routine
;
; At this point the current date is between equinox and Jan 1
; Days past equinox are AX - 80
PAST_EQUINOX:
        MOV     BX,80            ; Days from Jan 1 to equinox
        SUB     AX,BX            ; Subtract from days count
DAYS_DONE:
; AX holds number of days from spring equinox to present date
        MOV     EQNX_DAYS,AX     ; Store in variable
;                                |   ST(0)   |  ST(1)   |  ST(2)   |
        FLD     DEGS_PER_DAY     ; 0.9863.. |---------|
        FILD    EQNX_DAYS        ;    d      | 0.9863..|---------|
        FMULP   ST(1),ST         ;   degs    |---------|
; ST(0) now holds the number of degrees from spring equinox to
; current date
        RET
;
DATE_TO_DEG     ENDP
```

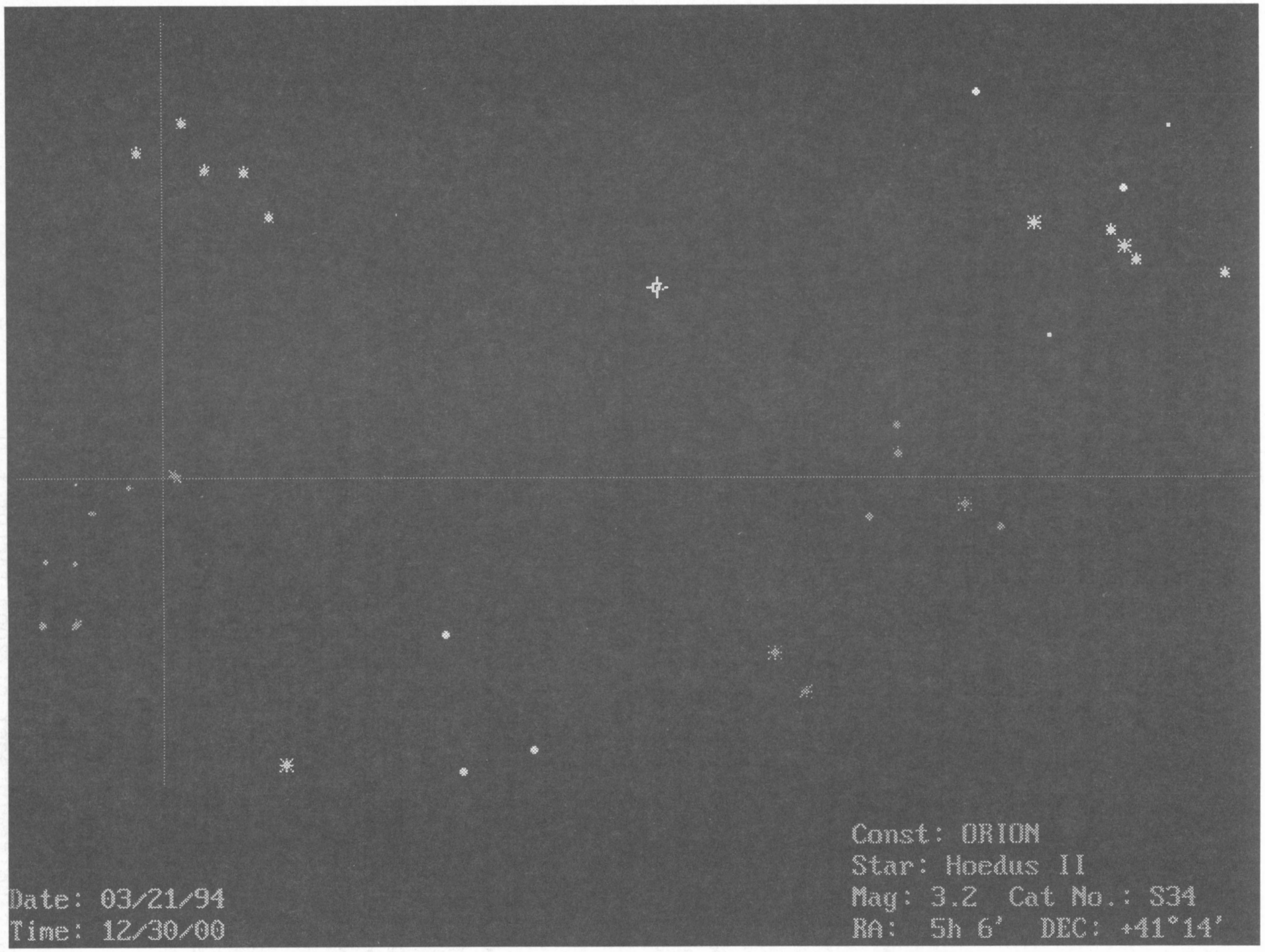
Once the program has obtained the decimal data for the current date and time, it can proceed to display each of the constellations in the database. Each constellation contains a set of entries for the celestial objects that it encodes. The constellation header contains the number of objects in the constellation, the attribute to be used at display time, and the number of the constellation. This last item can be used to index into a list of constellation names, as previously described. The procedure named SHOW_CONST, following, displays all objects in a constellation file.

```
;****************************************************************
;          code segment data for local procedures
;****************************************************************
;
OBJ_MAG         DB      0       ; Magnitude of current object
OBJECT_CNT      DB      0       ; Counter for number of objects
                                ; in constellation file
;
; Numeric constants
SIXTY           DQ      60.0    ; Divisor for minutes to decimal
FIFTEEN         DQ      15.0    ; Multiplier for hours to degrees
NINETY          DQ      90.0    ; DEC to pole distance
THREE_SIXTY     DQ      360.0   ; For complementing angles
TEN             DQ      10.0    ; For adjusting zoom range
;
; Object coordinates and transformation factors
CART_X          DQ      0       ; x cartesian coordinate
CART_Y          DQ      0       ; y cartesian coordinate
ZOOM_FACTOR     DQ      3.0     ; Enlargement multiplier
ZOOM_INTEGER    DW      0       ; Integer of ZOOM_FACTOR
ROT_ANGLE       DW      0       ; Rotation angle
SIN_@           DQ      0       ; Sine of rotation angle
COS_@           DQ      0       ; Cosine of rotation angle
; Displacement defaults
X_ORIGIN        DW      320     ; x origin pixel displacement
Y_ORIGIN        DW      240     ; y origin pixel displacement
;
; Display coordinates
OBJECT_X        DW      0       ; x screen coordinate of object
OBJECT_Y        DW      0       ; y screen coordinate of object
; Approximation controls
BEST_XY         DW      0       ; Best x+y approximation
;
;****************************************************************
;     procedure to display objects in a constellation file
;****************************************************************
SHOW_CONST      PROC    NEAR
; Display all objects in a constellation file according to
; magnitude
; On entry:
;         SI -- constellation file
;
;***************************|
;    reset DS to local data |
;***************************|
        MOV     CX,C0_DATA      ; Local segment
        MOV     DS,CX           ; To DS
        ASSUME  DS:C0_DATA      ; Assume this segment
;****************************|
;  multiple entry point      |
;****************************|
; Obtain constellation data
        MOV     AL,[SI+1]       ; Get number of objects
        MOV     CS:OBJECT_CNT,AL; Store in variable
        MOV     AL,[SI+2]       ; Get constellation color
        MOV     CONST_COLOR,AL  ; Store color in DS variable
        ADD     SI,4            ; Index to first star
NEXT_OBJECT:
        CALL    SHOW_OBJECT     ; Local procedure (listed
   ; below)
        DEC     CS:OBJECT_CNT   ; Decrement counter
        JZ      END_OF_OBJECTS  ; Go if at end
        ADD     SI,38           ; Index to next object
        JMP     NEXT_OBJECT     ; Continue
END_OF_OBJECTS:
        RET
SHOW_CONST      ENDP
```

The actual object display operations are performed by the procedure named SHOW_OBJECT, in the following listing. Processing is elaborate since the procedure first takes into account the current enlargement factor. This means that if the display operation is larger than the default (zoomed in), then the objects are displayed using a bitmap larger than would have normally been the case. By the same token, if the display is zoomed out, then the stars are displayed in a smaller magnitude. The object's coordinates are also adjusted to the current zoom factor.

The scaling transformation is performed according to the principles discussed in Chapter 3. The object is rotated and translated according to the stored parameters for these transformations. Rotation of coordinates is also performed by applying the standard formulas. This operation is executed by an auxiliary procedure named ROTATE. Another auxiliary procedure, called VIDEO_LIMITS, also listed below, checks that the final display coordinates are within the accepted range. Figure 11.7 is a second screen dump of the Astrium program in which the original display has been zoomed-in, rotated, and translated.

```
;****************************************************************
;             procedure to display celestial object
;****************************************************************
;
SHOW_OBJECT     PROC    NEAR
; Display celestial object according to object type and location
; On entry:
;       SI -- Start of object data block, formatted as follows:
; Celestial object data format:
;         OFFSET      UNIT      CONTENTS
;           0         byte      Star number in constellation
;                               (1-based)
;           1         byte      Object type:
;                                  1 = star
;                                  2 = star cluster
```


Const: ORION
Star: Hoedus II
Mag: 3.2  Cat No.: S34
RA:  5h 6'  DEC: +41°14'

Date: 03/21/94
Time: 12/30/00

Figure 11.7 *Zoom-in, Rotation, and Translation of the Screen Image in Figure 11.6*

```
;                                3 = nebula
;                                4 = galaxy
;       2-17        string     Star name (ASCII)
;        18          byte      Integer of magnitude (binary)
;        19          byte      Fraction of magnitude (binary)
;        20          word      RA hours (binary)
;        22          word      RA minutes (binary)
;        24          word      DEC degrees (binary)
;        26          word      DEC minutes (binary)
;       28-33       string     Star code (ASCII)
;        34          word      Storage for x screen coordinate
;        36          word      Storage for y screen coordinate
;       +38    ---------------  start of next object
;
;**************************|
;      get object type     |
;**************************|
        MOV     AL,[SI+1]       ; Get object type
        CMP     AL,1            ; 1 is star
        JE      STAR_TYPE
;***************************|
;  illegal object type      |
;***************************|
; No other object type is presently implemented
        STC                     ; Carry is error flag
        RET
;***************************|
;     star type object      |
;***************************|
STAR_TYPE:
        MOV     AL,[SI+18]      ; Get object magnitude
; Original magnitude is decreased or increased according to
; current enlargement factor (ZOOM_FACTOR), as follows:
; ZOOM_FACTOR = 2 to 4 then OBJ_MAG unchanged
;                4 then OBJ_MAG + 1
;                6 then OBJ_MAG + 2
;                2 to 1 then OBJ_MAG - 1
;                1 the OBJ_MAG - 2
; First round ZOOM_FACTOR to integer and add 10 to avoid
; negative zoom range
        FLD     CS:ZOOM_FACTOR  ; Factor ST(0)
        FADD    CS:TEN          ; Add 10 to zoom factor
        FRNDINT                 ; Round to integer
        FISTP   CS:ZOOM_INTEGER ; Stored as integer
        MOV     BX,CS:ZOOM_INTEGER      ; Zoom to BX
; BX now holds integer of zoom factor + 10
        CMP     BX,14           ; Test for upper limit
        JAE     BIGGER_STAR     ; Make image larger
        CMP     BX,12           ; Test lower limit
        JBE     SMALLER_STAR    ; Make image smaller
; At this point magnitude is not changed
```

```
        JMP     STORE_MAG
; Magnitude is decremented to make star image larger
BIGGER_STAR:
        CMP     BX,16           ; Test for magnitude  6
        JBE     MAG_PLUS_1      ; Increment 1 magnitude
        DEC     AL              ; Increment once
        JZ      BIG_LIMIT       ; Go if magnitude = 0
MAG_PLUS_1:
        DEC     AL              ; Increment magnitude
BIG_LIMIT:
        JMP     STORE_MAG
; Test for magnitude  1 for single reduction
SMALLER_STAR:
        CMP     BX,10           ; Test for 0 or less zoom
        JBE     MIN_MAG         ; Set minimum magnitude
        CMP     BX,11           ; Limit for double reduction
                                ; in magnitude
        JAE     MAG_MINUS_1     ; Go if 1 or less
        INC     AL              ; Reduce image
MAG_MINUS_1:
        INC     AL
        JMP     STORE_MAG       ; Save it
MIN_MAG:
        MOV     AL,6            ; Minimum magnitude
;***************************|
;  store adjusted magnitude |
;***************************|
STORE_MAG:
        MOV     CS:OBJ_MAG,AL   ; Store magnitude
;***************************|
;  compute RA at equinox    |
;***************************|
        FILD    WORD PTR[SI+20] ; Load RA hours
;          .                    |   ST(0)   |  ST(1)   |  ST(2)   |
                                ; RA hours |---------|
        FILD    WORD PTR[SI+22] ; RA min    | RA deg   |----------|
        FDIV    CS:SIXTY        ; RA min/60| RA hours|----------|
        FADD                    ; RAh+RAm   |---------|
        FMUL    CS:FIFTEEN      ;     * 15  |---------|
                                ; = RA deg |
        FILD    WORD PTR[SI+24] ; DEC deg   |   RA     |----------|
        FILD    WORD PTR[SI+26] ; DEC min   | DEC deg |    RA    |
        FDIV    CS:SIXTY        ;DEC min/60| DEC deg |    RA    |
        FADD                    ;   DEC     |   RA     |----------|
        FLD     CS:NINETY       ;   90      |   DEC    |    RA    |
        FXCH                    ;   DEC     |   90     |    RA    |
        FSUB                    ; 90 - DEC |   RA     |----------|
        FXCH                    ;    RA     | DEC - 90|----------|
        CALL    DEG_TO_RAD      ; RA rads   | DEC - 90|----------|
        FLD     ST(0)           ; RA rads   | RA rads | 90 - DEC|
        CALL    SINE            ; sin RA    | RA rads | 90 - DEC|
        FMUL    ST,ST(2)        ;    x     | RA rads | 90 - DEC|
        FCHS                    ; x * -1   | ------- |
        FSTP    CS:CART_X       ; RA rads  | 90 - DEC| ------- |
        CALL    COSINE          ; cos RA   | 90 - DEC| ------- |
        FMULP   ST(1),ST        ;    y     | ------- |
        FSTP    CS:CART_Y       ; -------- |
; At this point the signed cartesian coordinates of the object
; are stored in CS:CART_X and CS:CART_Y
;***************************|
;       scale object        |
;***************************|
        FLD     CS:CART_X       ;    x     | ------- |
        FLD     CS:ZOOM_FACTOR  ;    ?     |    x    | ------- |
        FMULP   ST(1),ST        ; x * fac  | ------- | ------- |
        FSTP    CS:CART_X       ; -------- |
; Process y
        FLD     CS:CART_Y       ;    y     | ------- |
        FLD     CS:ZOOM_FACTOR  ;    ?     |    y    | ------- |
        FMULP   ST(1),ST        ; y * fac  | ------- | ------- |
        FSTP    CS:CART_Y       ; -------- |
;***************************|
;       rotate object       |
;***************************|
        CALL    ROTATE
;***************************|
;      translate object     |
;      into video plane     |
;***************************|
; y coordinate
        FLD     CS:CART_Y       ;    y     | ------- |
        FILD    CS:Y_ORIGIN     ;  240     | y * fac | ------- |
        FADD                    ; 240 + y  | ------- | ------- |
        FISTP   CS:OBJECT_Y     ; -------- |
; x coordinate
        FLD     CS:CART_X       ;    x     | ------- |
        FILD    CS:X_ORIGIN     ;  320     | x * fac | ------- |
        FADD                    ; 320 + x  | ------- | ------- |
        FISTP   CS:OBJECT_X     ; -------- |
; Test for negative coordinates
        MOV     BX,CS:OBJECT_X  ; BX = x screen coordinate
        MOV     CX,CS:OBJECT_Y  ; CX = y screen coordinate
        TEST    BX,8000H        ; Is high bit set?
        JNZ     NEG_COORD       ; Go if set
        TEST    CX,8000H        ; Same for y coordinate
        JNZ     NEG_COORD       ; Go if negative
; At this point coordinates are positive
        JMP     POS_COORDS      ; Go
NEG_COORD:
; Reset variables if coordinates are negative
        MOV     CS:OBJECT_X,0FFFFH      ; Indicates invalid
        MOV     CS:OBJECT_Y,0FFFFH
        JMP     BAD_OBJ_COORDS  ; Quick exit from routine
; Object is ready to display
POS_COORDS:
        PUSH    SI              ; Save object pointer
;***************************|
;     select star bitmap    |
;***************************|
; Object's integer magnitude is used to select star bitmap
        MOV     AL,CS:OBJ_MAG   ; Magnitude to AL
        CMP     AL,0            ; Test magnitude
        JNE     TEST_MAG1       ; Go if not zero
; Display object of magnitude 0
        MOV     MAG0_X,BX       ; Coordinates to display block
        MOV     MAG0_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,4            ; y displacement
        LEA     SI,MAG0_X       ; Pointer to bitmap block
        JMP     OBJECT_DISPLAY  ; Go to display routine
TEST_MAG1:
        CMP     AL,1            ; Test magnitude
        JNE     TEST_MAG2       ; Go if not zero
; Display object of magnitude 1
        MOV     MAG1_X,BX       ; Coordinates to display block
        MOV     MAG1_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,3            ; y displacement
        LEA     SI,MAG1_X       ; Pointer to bitmap block
        JMP     OBJECT_DISPLAY  ; Go to display routine
TEST_MAG2:
        CMP     AL,2            ; Test magnitude
        JNE     TEST_MAG3       ; Go if not zero
; Display object of magnitude 2
        MOV     MAG2_X,BX       ; Coordinates to display block
        MOV     MAG2_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,2            ; y displacement
        LEA     SI,MAG2_X       ; Pointer to bitmap block
        JMP     OBJECT_DISPLAY  ; Go to display routine
TEST_MAG3:
        CMP     AL,3            ; Test magnitude
        JNE     TEST_MAG4       ; Go if not zero
; Display object of magnitude 3
        MOV     MAG3_X,BX       ; Coordinates to display block
        MOV     MAG3_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,1            ; y displacement
        LEA     SI,MAG3_X       ; Pointer to bitmap block
        JMP     OBJECT_DISPLAY  ; Go to display routine
TEST_MAG4:
        CMP     AL,4            ; Test magnitude
        JNE     TEST_MAG5       ; Go if not zero
```

```
; Display object of magnitude 4
        MOV     MAG4_X,BX        ; Coordinates to display block
        MOV     MAG4_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,0             ; y displacement
        LEA     SI,MAG4_X        ; Pointer to bitmap block
        JMP     OBJECT_DISPLAY   ; Go to display routine
TEST_MAG5:
; Display object of magnitude 5
        MOV     MAG5_X,BX        ; Coordinates to display block
        MOV     MAG5_Y,CX
; y axis displacement of object center according to magnitude
        MOV     AX,0             ; y displacement
        LEA     SI,MAG5_X        ; Pointer to bitmap block
;***************************|
;       display object      |
;***************************|
OBJECT_DISPLAY:
; Adjust and store object coordinates in memory
        ADD     BX,5             ; Add x displacement
        ADD     CX,AX            ; and y displacement
        CALL    VIDEO_LIMITS     ; Local procedure to test valid
                                 ; coordinate range
        JNC     DO_VIDEO         ; Go if carry clear
; At this point the object has coordinates outside the legal range
; Skip display function and enter invalid coordinates codes
        POP     SI               ; Restore object pointer
        MOV     WORD PTR [SI+34],0FFFFH ; x is invalid
        MOV     WORD PTR [SI+36],0FFFFH ; y is invalid
BAD_OBJ_COORDS:
        RET
; Save object coordinates and display object
DO_VIDEO:
        PUSH    BX               ; Save x/y in stack
        PUSH    CX
        LEA     BX,CONST_COLOR   ; Color of star
        CALL    MONO_MAP_18      ; Library procedure
        POP     CX               ; Restore x and y coordinates
        POP     BX               ; of object
; Store object coordinates in database
        POP     SI               ; Restore object pointer
        MOV     WORD PTR [SI+34],BX     ; x coordinate
        MOV     WORD PTR [SI+36],CX     ; y coordinate
        RET
SHOW_OBJECT     ENDP
;*****************************************************************
;            limits of the video display area
;*****************************************************************
VIDEO_X_RT      EQU     630     ; Right screen limit
VIDEO_X_LF      EQU       4     ; Left screen limit
VIDEO_Y_DN      EQU     400     ; Lower screen limit
```

In the alpha modes the mouse driver manages the text cursor on a coarse grid of screen columns and rows, according to the active display mode. VGA programs that execute in graphics modes must provide their own cursor bitmap, which is installed by means of an interrupt 33H subservice. However, since the graphics cursor operated by the driver is limited to a size of 16-by-16 pixels, many graphics programs create and manage their own cursor. In this case the driver services are used to detect mouse movements, but the actual cursor operation and display are handled directly by the application. This is also the case of XGA programs that use the sprite functions to manage a mouse cursor image. The implementation of a cursor in a VGA graphics mode is discussed later in this chapter.

In addition to mouse cursor management and display, the subservices of interrupt 33H include functions to set the mouse sensitivity and rate, to read button press information, to select video pages, and to initialize and install interrupt handlers that take control when the mouse is moved or when the mouse buttons are operated. However, some of the services in the interrupt 33H drivers reprogram the video hardware in ways that can conflict with an application. For this reason, we have limited our discussion to those mouse services that are not directly related to the video environment. These services can be used from any VGA, XGA, or SuperVGA graphics modes without interference. However, in this case, it is the application's responsibility to perform all video updates.

## Subservice 0 — Initialize Mouse

Subservice number 0 of interrupt 33H is used to reset the mouse device and to obtain its status. An application usually calls this service to certify that the mouse driver is resident and to initialize the device parameters. The following fragment shows a call to this subservice:

```
; Initialize mouse by calling subservice 0 of interrupt 33H
        MOV     AX,0            ; Reset mouse hardware and
                                ; software
        INT     33H             ; Mouse interrupt
        CMP     AX,0            ; Test for error during reset
        JNZ     OK_RESET        ; No problem
; At this point the program should provide an error routine to
; handle an invalid initialization call
        .
        .
        .
; Execution continues at this label if the mouse was initialized
OK_RESET:
        .
        .
        .
```

### 13.3.2 Checking Mouse Installation

Applications that use the mouse device must adopt one of three alternatives regarding the support software: assume that the driver was installed by the user, load a driver program, or provide the low-level services within its code. By far, most applications adopt the first option, that is, assume that the user has previously loaded the mouse driver software, although the more refined programs that use a mouse device include an installation utility that selects the appropriate driver and creates or modifies a batch file in order to ensure that the mouse driver is resident at the time of program execution.

In any case, the first operation usually performed by an application that plans to use the mouse control services in interrupt 33H is to test the successful installation of the driver program. Since the driver is vectored to interrupt 33H, this test consists simply of checking that the corresponding slot in the vector table is not a null value (0000:0000H) or an IRET operation code. Either one of these alternatives indicates that no mouse driver is available. The following fragment shows the required processing:

```
; Code to check if mouse driver software is installed in the
; interrupt 33H vector. The check is performed by reading the
; interrupt 33H vector using MS-DOS service number 53,
; of INT 21H
        MOV     AH,53           ; MS-DOS service request
        MOV     AL,33H          ; Desired interrupt number
        INT     21H             ; MS-DOS service
; ES:BX holds address of interrupt handler, if installed
        MOV     AX,ES           ; Segment to AX
        OR      AX,BX           ; OR with offset
        JNZ     OK_INT33        ; Go if not zero
; Test for an IRET opcode in the vector
        CMP     BYTE PTR ES:[BX],0CFH    ; CFH is IRET opcode
        JNE     OK_INT33        ; Go if not IRET
; At this point the program should provide an error handler
; to exit execution or to load a mouse driver
        .
        .
        .
; Execution continues at this label if a valid address was found
; in the interrupt 33H vector
OK_INT33:
        .
        .
        .
```

### 13.3.3 Subservices of Interrupt 33H

The Microsoft mouse interface was designed to provide control of the mouse device from high- and low-level languages. VGA alphanumeric programs can use the Microsoft mouse software by selecting one of two available text cursors.

One or more characters can be entered directly into the buffer so they appear to an application as if they had been typed from the keyboard. In this case the buffer pointers also require adjustment, since they must signal both ends of the new string. One possible use of this technique is to force DOS to execute a program or command on exit from another program.

## 13.3 Programming the Mouse

The IBM *Personal System/2 and Personal Computer BIOS Interface Technical Reference* (see Bibliography) describes a pointing device interface associated with service number 194 of INT 15H. However, there are several difficulties associated with this service. In the first place, the IBM documentation dealing with this mouse service is not sufficient for programming the device. Another consideration is that the services are not compatible with different mouse hardware. Then there is the problem that various non-IBM versions of the BIOS do not include this service. Finally, the service is not recognized in the DOS mode of OS/2.

If the BIOS mouse services of INT 15H were operational and compatible with standard mouse hardware, a program could use these functions much the same way as it uses the video, printer, or communications services in the BIOS. Due to the difficulties mentioned in the preceding paragraph, most applications must find alternative ways of controlling mouse operation. However, all alternative solutions have the disadvantage of requiring an installed mouse driver. This leaves three alternatives: the software must assume that the user has previously installed and loaded a compatible mouse driver, the software must provide an installation routine that loads the driver, or the code must include a low-level driver for the mouse device.

### 13.3.1 The Microsoft Mouse Interface

The mouse driver software that has achieved general acceptance is the one by Microsoft Corporation. The Microsoft mouse control software is installed as a system driver or as a TSR program. The system version is usually stored in a disk file with the extension .SYS and the TSR version in a file with the extension .COM. The Microsoft mouse interface services are documented in the book *Microsoft Mouse Programmer's Reference*, published by Microsoft Press (see Bibliography).

Most manufacturers of mouse devices provide drivers that are compatible with the one by Microsoft. Therefore, the use of the Microsoft mouse interface is not limited to mouse devices manufactured by this company, but extends to all Microsoft-compatible hardware and software. The installation command for the mouse driver is usually included in the CONFIG.SYS or AUTOEXEC.BAT files. The Microsoft mouse interface attaches itself to software interrupt 33H and provides a set of 36 subservices. These mouse subservices are accessible by means of an INT 33H instruction.

### 13.2.3 Keyboard Status Bytes

Figure 13.1 lists the bit structure of the first keyboard status byte located at address 0040:0017H. Figure 13.2 lists the bit structure of the second keyboard status byte located at address 0040:0018H.

Bits 4, 5, 6, and 7 on both status bytes refer to the Scroll Lock, Num Lock, Caps Lock, and Ins keys. In the first status byte the bits indicate the active state. In the second status byte they indicate if these keys were pressed simultaneously with the last keystroke. Bits 0 and 1 of the second status byte are meaningful for PS/2 systems only, since previous keyboards do not have Left Alt and Left Ctrl keys. Bit 2 of this byte is also used, in the PCjr, to indicate the state of the keyboard click function.

An application can change the alternate state of the toggle keys by modifying the corresponding bit in the keyboard status bytes. This operation can be used for presetting the lock key status. In the AT and PS/2 keyboards the state of the toggle keys is represented by an illuminated indicator (LED). Changing the corresponding bit to 1 or 0 turns the light on or off.

### 13.2.4 Keyboard Buffers

The keyboard buffer is used by the interrupt 09H handler to store the ASCII values for keystrokes. Applications and system programs can retrieve these values using the BIOS keyboard services of INT 16H or by reading the buffer directly. The handler at INT 09H converts the original scan codes into the corresponding ASCII, extended ASCII, and control codes of the IBM character set, prior to storing the keystrokes in the buffer.

The keyboard buffer uses two buffer pointers. The output pointer, called BUFFER_HEAD, is located at address 0040:001AH. The input pointer, called BUFFER_TAIL, is located at address 0040:001CH. The storage area of the buffer starts at address 0040:001EH and extends for 16 words. Two storage bytes are used for each keystroke, one for the ASCII value and another one for the scan code. Since the last word in the buffer is not used for character storage, the total capacity is 15 characters.

If the keyboard buffer is empty, then the BUFFER_HEAD pointer and the BUFFER_TAIL pointer are equal. This can be used to flush any old characters from the buffer. The following code fragment shows a way of flushing the keyboard buffer:

```
        CLI                       ; Interrupts OFF while changing
                                  ; pointers
        MOV     AX,0040H          ; BIOS data area segment
        MOV     ES,AX             ; indirectly to ES
        MOV     AL,ES:[001AH]     ; Get BUFFER_HEAD
        MOV     ES:[001CH],AL     ; Set BUFFER_TAIL to this value
        STI                       ; Interrupts back on
        .
        .
        .
```

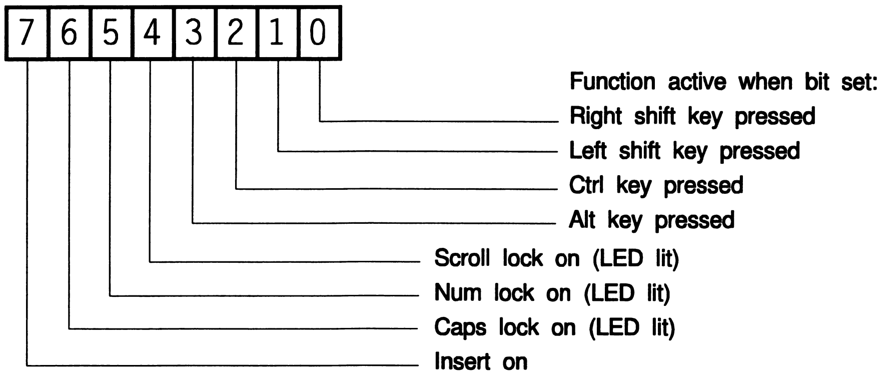


Figure 13.1 *First Keyboard Status Byte (0040:0017H)*

keyboard buffer for the special keystrokes, but the execution of an application cannot be interrupted by pressing Ctrl-C.

### 13.2.2 Keyboard Data in BIOS

Keyboard data is stored in three BIOS areas:

1. The byte at absolute address 0040:0017H, known as the first keyboard status byte
2. The byte at absolute address 0040:0018H, known as the second keyboard status byte
3. The buffer pointers and keyboard data buffer starting at absolute address 0040:001AH

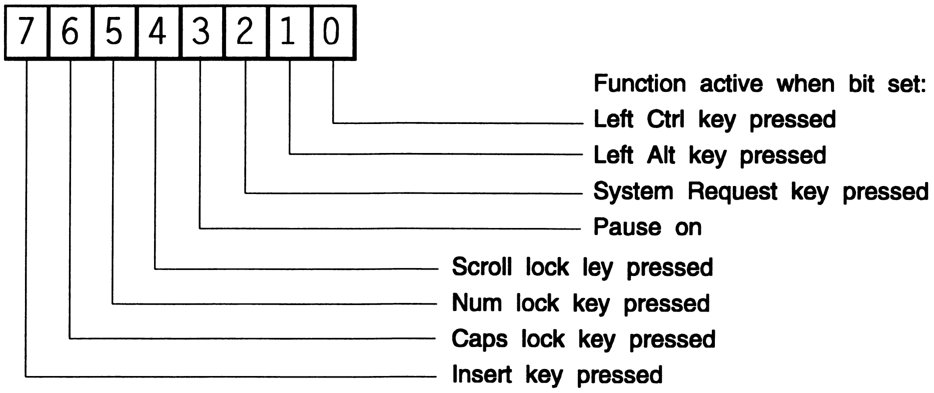


Figure 13.2 *Second Keyboard Status Byte (0040:0018H)*

in the original PC, the XT, or the PCjr, since the processor used in these computers does not support multitasking.

The AT and the PS/2 systems, upon detecting the system request key or keystroke combinations, loads the AH registers with the value 85H and executes INT 15H. If the action was caused by a make code, the AL register holds 00H. If the action was caused by a break code, AL holds 01H. Since BIOS does not provide a function for service number 85H, of INT 15H, the keystroke normally appears to have no action, but an application could take over this service for its own purposes.

## Pause Function

All IBM microcomputers go into a wait loop every time that the Pause key or keystroke combination is detected by the interrupt 09H handler. This allows the user to instantly detain an application or system function, such as a screen listing or printer operation. The paused program resumes when an ASCII key is pressed. In the PS/2 keyboards this is a dedicated key labeled Pause. In the PC, XT, and AT keyboards the pause function activates with the Ctrl-Num Lock sequence. In the PCjr the pause function requires the Fn-Q keystroke.

## Break Function

The BIOS keyboard handler in all the IBM microcomputers recognizes certain key combinations as a keyboard break. This function is provided so that a system program or an application can regain control of run away code, interrupt undesired execution, or exit an endless loop. This is possible because the function is triggered by a hardware interrupt physically linked to the key or keystroke combination designated as the keyboard break. If the keyboard interrupt is enabled, this action always invokes interrupt 1BH.

When the break handler concludes, it should proceed in the same manner as any hardware interrupt routine. The code should send the End of Interrupt command to the interrupt controller, reenable interrupts, and exit via the IRET instruction, for example:

```
MOV     AL,20H          ; Issue EOI command
OUT     20H,AL          ; to interrupt controller
STI                     ; Interrupts ON
IRET                    ; Return for interrupt
```

## Ctrl-C Handler

MS- DOS provides its own break handler which is activated with the keystrokes Ctrl-C. The DOS routine terminates any currently active process and returns control to the parent process. This constitutes an actual abort operation, but it does not close any open files nor restore vectors or drivers. The most important difference between the DOS Ctrl-C handler and the keyboard break is that the DOS handler is not interrupt driven. When DOS is in control, it checks the

2. The software control keys are noncharacter keys used by system and applications programs for executing user controls and commands. This group includes the function keys, labeled F1 to F12, the Ins, Insert, Del, Delete, Home, End, Page-Up, Page-Dn, Page-Down, Tab, Backspace, Esc, and the arrow keys.
3. The hot keys and keystroke combinations are those that generate an immediate action at the system or application level. The Pause, Print Screen, Ctrl-Alt-Del, and SysRq actions correspond to system level functions. The Ctrl-Break sequence is available as a programmable hot keystroke combination.
4. The alternate state keys are those that temporarily activate an alternative interpretation of another key. They are the Shift, the Ctrl, and the Alt keys.
5. The toggle keys are those that permanently activate one of two states, which are recorded at the system level. Caps Lock, Num Lock, and Scroll Lock are toggle keys.

When the keyboard handler detects a scan code corresponding to an ASCII or a software control key (groups 1 and 2 above), it looks up the code corresponding to the key and stores it in an area of RAM called the keyboard buffer.

The hot keys (group 3) determine an immediate action either in the form of a routine internal to the keyboard handler or associated with a software interrupt. Some hot keys consist of several keystrokes; for instance, the Ctrl-Alt-Del sequence activates the warm boot action and the Ctrl-Break sequence transfers control to interrupt 1BH.

Some alternate state keys (group 4) determine a different interpretation of a keystroke by the handler; for instance, if a character key is pressed while the Shift key is held down, the keystroke is recorded in upper case. Other alternate state keys are merely recorded by the handler and the interpretation is left to the application.

This action applies also to the toggle keys (group 5). On the PC AT and the PS/2 keyboards each toggle key is furnished with an indicator light, sometimes called a keyboard LED. In these keyboards, the indicator light is also controlled by the handler at INT 09H.

### Print Screen Function

All IBM microcomputers provide a hot key or keystroke combination that activates a screen dump to the parallel printer port. The BIOS handler for this function is located at the vector for INT 05H.

### System Request Function

The system request key, labeled SysRq, is found at the top, right-hand corner of the AT keyboard. This key was conceived for use in multitasking environments, but the developers of OS/2 adopted the Ctrl-Esc and the Alt-Esc keystrokes instead. In the PS/2 keyboards the system request function, labeled SysRq, requires the Alt-Print Screen keystroke combination. No system request function is implemented

Table 13.1 *Keyboard Make and Break Scan Codes*

| KEY | PC & XT | AT | PCJR | PS/2 |
|---|---|---|---|---|
| `<Esc>` | 01H-81H | 01H-81H | 01H-81H | 01H-81H |
| `<F1>` | 3BH-BBH | 3BH-BBH | 3BH-BBH | 3BH-BBH |
| `<F2>` | 3CH-BCH | 3CH-BCH | 3CH-BCH | 3CH-BCH |
| `<F3>` | 3DH-BDH | 3DH-BDH | 3DH-BDH | 3DH-BDH |
| `<F4>` | 3EH-BEH | 3EH-BEH | 3EH-BEH | 3EH-BEH |
| `<F5>` | 3FH-BFH | 3FH-BFH | 3FH-BFH | 3FH-BFH |
| `<F6>` | 40H-C0H | 40H-C0H | 40H-C0H | 40H-C0H |
| `<F7>` | 41H-C1H | 41H-C1H | 41H-C1H | 41H-C1H |
| `<F8>` | 42H-C2H | 42H-C2H | 42H-C2H | 42H-C2H |
| `<F9>` | 43H-C3H | 43H-C3H | 43H-C3H | 43H-C3H |
| `<F10>` | 44H-C4H | 44H-C4H | 44H-C4H | 44H-C4H |
| `<Scroll Lock>` | 46H-C6H | 46H-C6H | 46H-C6H | 46H-C6H |
| `<Backspace>` | 0EH-8EH | 0EH-8EH | 0EH-8EH | 0EH-8EH |
| `<Ins>` | 52H-D2H | 52H-D2H | 52H-D2H | 52H-D2H |
| `<Del>` | 53H-D3H | 53H-D3H | 53H-D3H | 53H-D3H |
| `<Num Lock>` | 45H-C5H | 45H-C5H | 45H-C5H | 45H-C5H |
| `<Tab>` | 0FH-8FH | 0FH-8FH | 0FH-8FH | 0FH-8FH |
| `<Enter>` | 1CH-9CH | 1CH-9CH | 1CH-9CH | 1CH-9CH |
| `<Caps Lock>` | 3AH-BAH | 3AH-BAH | 3AH-BAH | 3AH-BAH |
| `<Left Shift>` | 2AH-AAH | 2AH-AAH | 2AH-AAH | 2AH-AAH |
| `<Right Shift>` | 36H-B6H | 36H-B6H | 36H-B6H | 36H-B6H |
| `<Left Ctrl>` | 1DH-9DH | 1DH-9DH | 1DH-9DH | 1DH-9DH |
| `<Up Arrow>` | 48H-C8H | 48H-C8H | 48H-C8H | 48H-C8H |
| `<Left Arrow>` | 4BH-CBH | 4BH-CBH | 4BH-CBH | 4BH-CBH |
| `<Right Arrow>` | 4DH-CDH | 4DH-CDH | 4DH-CDH | 4DH-CDH |
| `<Down Arrow>` | 50H-D0H | 50H-D0H | 50H-D0H | 50H-D0H |

1. If the scan code in port 60H corresponds to a character key, the handler places the corresponding ASCII code, together with the scan code, in the keyboard buffer.
2. If the scan code corresponds to the make or break action of the Shift, Ctrl, Alt, Ins, Num Lock, Cap Lock, Scroll Lock, or the SysRq key, the handler updates the state of the corresponding bit or bits in a memory area known as the keyboard status byte.
3. If the scan code corresponds to the Del key, and if the Ctrl-Alt keys are being simultaneously held down, the handler transfers execution to the BIOS warm boot routine.
4. If the scan code for the Pause key, or an equivalent sequence, is detected, the handler enters a wait loop until the next valid keystroke is received.
5. If the scan code corresponds to the Print Screen key or key combination, the handler executes interrupt 05H.
6. If the scan code corresponds to the Ctrl Break sequence, the handler executes interrupt 1BH.

### 13.2.1 Classification of Keys and Keystrokes

According to their function, the keys can be classified into the following groups:

1. The character keys are those that are commonly used in the creation of text, namely, the letters of the alphabet, the keys for the Arabic numerals, and the symbol keys. These keys are often called the ASCII set.

Although the two chips are different, the 8042 can be programmed to mimic the 8048.

### 13.1.1 Keyboard Controller

The main function of the keyboard controller is to relieve the microprocessor from monitoring the state of the key switches. In operation these chips perform as follows:

1. Every time that a key is pressed or released, the 8048 or 8042 stores a code (called a scan code) in one of its internal registers. In all PC systems this register can be read at port 60H. The scan codes are specific for each key but do not correspond to the ASCII value of the key.
2. Once the scan code is stored, the keyboard controller generates an interrupt on the 8259 line IRQ1. If the keyboard interrupt is enabled, the microprocessor transfers execution to the INT 09H handler.
3. The INT 09H handler, located in the system BIOS, reads the scan code at port 60H and converts it into the ASCII or extended ASCII characters. If the keystroke corresponds to a key that requires immediate action, as is the case with the LOCK keys, the Print Screen key, or any other hot key, the handler proceeds accordingly. If not, the ASCII code and the original scan code are placed in a BIOS area called the keyboard buffer.

### 13.1.2 The Keyboard Scan Codes

Some forms of keyboard programming require the identification of keys by their scan codes. Every time a key is pressed or released, the keyboard microprocessor places a code, particular to that key, in the register associated with port 60H. The keyboard handler uses these scan codes to determine which key or key combination has been pressed or released and to take action accordingly.

The programming operations necessary for obtaining and interpreting the keyboard scan codes are simple and straightforward. In addition, IBM has maintained the same scan codes for the corresponding keys in the various keyboards.

The keyboards of the PS/2 line offer three sets of scan codes. Scan code set 1 is compatible with the keyboards of the PC line. Table 13.1 shows the make and break scan codes for some common control keys, which are identical in all PC keyboards.

## 13.2 Keyboard Programming

The IBM keyboard controllers are normally programmed to generate an interrupt every time a make or break scan code is placed in port 60H. The BIOS keyboard handler, located at the vector for INT 09H, gains control during this interrupt. The action performed by the handler depends on the key or key combination that originated the interrupt:

Chapter

# 13 Interactive Animation

## 13.0 User-Animated Objects

Interactive animation refers to screen objects that are moved at will by the user. Typically, the animated screen object is controlled by means of an input device, such as the keyboard, mouse, puck, or graphics tablet. Of all input devices two have gained general acceptance, as well as a certain level of software standardization: the keyboard and the mouse. In this chapter we discuss programming these devices as a means for animating interactive screen objects. Other interactive input devices, although often sophisticated and effective, are specialty tools that lie outside the scope of a general-purpose book.

## 13.1 PC Keyboard Hardware

The keyboard is one of IBM microcomputer components that has undergone the most modifications and redesigns. Externally, the changes have consisted in the addition of several new keys, repositioning of other keys, and the inclusion of light-emitting diodes to indicate the state of the toggle keys. Internally, the keyboards of the PC and XT, the one on the PCjr, and the keyboards of the PC AT and the PS/2 lines are completely different.

The keyboard furnished with the original IBM PC and PC XT has 83 keys. The IBM PCjr was released with a 62-key keyboard, popularly dubbed the "chiclet" keyboard due to the appearance and feel of the keys. It was soon recalled by IBM and replaced with a better model. The IBM PC AT has an 84-key keyboard characterized by a relocated Esc (escape) key and a new system request key, intended for use in multitasking environments. The PS/2 line introduced yet another keyboard with 101 keys on the U.S. version and 102 keys for the models sold outside the United States. The PS/2 keyboards are equipped with two additional function keys and with special keypads with duplicate editing and cursor controls keys.

Regarding electronic hardware, the PC and XT keyboards use the Intel 8048 keyboard controller while the AT and PS/2 keyboards use the Intel 8042.

```
        CMP     AH,AL             ; Compare current with final DAC
        JE      NO_BLU_ACTION     ; Go if equal
        INC     AL                ; Lighten value by one unit
        MOV     CS:[SI],AL        ; Store value
NO_BLU_ACTION:
        CALL    TIME_VR_B0
        OUT     DX,AL             ; Write DAC blue
        INC     SI                ; Bump pointers
        INC     DI
        LOOP    LIGHTEN_DAC       ; Continue
;**************************|
;     insert delay period   |
;**************************|
        PUSH    BX                ; Save iteration counter
        MOV     BX,CS:TIME_DELAY          ; Stored parameter
        CMP     BX,0              ; Test for no delay
        JE      NO_IN_DELAY       ; Go if no delay
        CALL    MILLI_TIME        ; Delay procedure
NO_IN_DELAY:
        POP     BX                ; Restore counter
; AT this point all DAC registers have been lightened by one color
; unit
        POP     DI                ; Restore caller's DAC pointer
        POP     SI                ; and work area pointer
        DEC     BX                ; BX is iteration counter
        JNZ     LIGHTEN_256
;*********************|
;   restore caller's  |
;       context       |
;*********************|
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
FADE_IN         ENDP
```

```
        LEA     SI,CS:WORK_DAC  ; Pointer to DAC storage area
        MOV     AL,0            ; DAC value for initialization
        PUSH    SI              ; Save pointer
CLEAR_WORK_DAC:
        MOV     CS:[SI],AL      ; Store 0H code
        INC     SI              ; Bump pointer
        LOOP    CLEAR_WORK_DAC
        POP     SI              ; Restore pointer
;***************************|
;   read table and set DAC  |
;***************************|
; Code maintains the current setting of the DAC register in
; a working storage table. This value is used to reset the DAC
; register to a shade closer to the final DAC register value
; First select DAC write operation
LIGHTEN_256:
        CALL    TIME_VR_B3
        MOV     CX,256          ; Counter for 256 DAC registers
        MOV     DX,03C8H        ; Pallet Address (Write mode)
        MOV     AL,0            ; Value to write
        OUT     DX,AL           ; To DAC Palette Address register
        INC     DX              ; 3C9H is DAC data register
        PUSH    SI              ; Save working area pointer
        PUSH    DI              ; and user DAC table pointer
;
LIGHTEN_DAC:
        MOV     AL,CS:[SI]      ; Load current DAC
        MOV     AH,[DI]         ; Get final DAC value
        CMP     AH,AL           ; Compare current with final DAC
        JE      NO_RED_ACTION   ; Go if equal
        INC     AL              ; Lighten value by one unit
        MOV     CS:[SI],AL      ; Store value
NO_RED_ACTION:
        CALL    TIME_VR_B0
        OUT     DX,AL           ; Write DAC red
        INC     SI              ; Bump pointers
        INC     DI
;
        MOV     AL,CS:[SI]      ; Load current DAC
        MOV     AH,[DI]         ; Get final DAC value
        CMP     AH,AL           ; Compare current with final DAC
        JE      NO_GRN_ACTION   ; Go if equal
        INC     AL              ; Lighten value by one unit
        MOV     CS:[SI],AL      ; Store value
NO_GRN_ACTION:
        CALL    TIME_VR_B0
        OUT     DX,AL           ; Write DAC green
        INC     SI              ; Bump pointers
        INC     DI
        MOV     AL,CS:[SI]      ; Load current DAC
        MOV     AH,[DI]         ; Get final DAC value
```

In the following procedure the fade-in can be slowed down by introducing a delay between each of the 64 incremental steps. The timing routine (called MILLI_TIME) was listed in Section 7.7.2.

```
;****************************************************************
;                   image fade-in procedure
;****************************************************************
;
FADE_IN         PROC    NEAR
; Gradually change all DAC registers from black to values in table
; furnished by caller
; On entry:
;       DS:DI -> caller's DAC register table (256 entries)
;         AX = milliseconds of time delay between palette
;              register groups
;
; Note: this procedure assumes that all DAC registers are black
;       (00H) on entry
;
; Logic:
;      The DAC registers are lightened from black to the caller's
;      desired values in 64 steps
;      In each iteration the code reads and stores the current
;      setting of the DAC registers
;      This setting is compared to the final DAC values passed
;      by the caller
;      If present DAC = caller's table then NO ACTION
;      If present DAC darker than caller's table then LIGHTEN DAC
;***************************|
;     store delay value     |
;***************************|
        MOV     CS:TIME_DELAY,AX        ; To CS variable
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
; Set counter for 64 fade iterations
        MOV     BX,63           ; BX holds fade iterations
;***************************|
;   prepare work buffer     |
;***************************|
; The work area for DAC values CS:WORK_DAC is first initialized
; to all black values
        MOV     CX,768          ; Values to clear
        INC     SI                ; Bump pointer
        MOV     AL,CS:[SI]        ; Load table value
        CMP     AL,0              ; Is register already black?
        JZ      DARKEN_BLUE       ; Go if already black
        DEC     AL                ; Darken one unit if not black
        MOV     CS:[SI],AL        ; Store value
DARKEN_BLUE:
        OUT     DX,AL             ; Write DAC red
        INC     SI                ; Bump pointer
        LOOP    DARKEN_DAC        ; Continue
;**************************|
;     insert delay period   |
;**************************|
        PUSH    BX                ; Save iteration counter
        MOV     BX,CS:TIME_DELAY          ; Stored parameter
        CMP     BX,0              ; Test for no delay
        JE      NO_CALLER_DELAY ; Go if no delay
        CALL    MILLI_TIME        ; Delay procedure
NO_CALLER_DELAY:
        POP     BX                ; Restore counter
; AT this point all DAC registers have been darkened by one color
; unit
        POP     SI                ; Restore DAC table pointer
        DEC     BX                ; BX is iteration counter
        JNZ     DARKEN_256
;
;*********************|
;   restore caller's   |
;       context        |
;*********************|
        POP     DI
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
FADE_OUT        ENDP
```

## 12.3.4 Color-Shift Fade-In

A fade-in takes place when the graphics image is gradually displayed, usually starting from a black or monochrome screen. In the VGA environment the fade-in transformation requires a table of values, furnished by the caller, which contains the final setting desired for the DAC registers. The fade-in consists of incrementing the color attribute in each DAC register until the corresponding table value is reached. Since each VGA DAC register has a value range from 0 to 63, the fade-in operation is performed in 64 steps. In XGA-2, with a color range of 0 to 255, the same fade can be done in 256 steps.

```
        INC     DX
        INC     DX              ; 3C9H is DAC data register
        LEA     DI,CS:WORK_DAC  ; Pointer to storage area
READ_DAC:
        IN      AL,DX           ; Read DAC red
        MOV     CS:[DI],AL      ; Store in RAM
        INC     DI              ; Bump pointer
        IN      AL,DX           ; Read DAC green
        MOV     CS:[DI],AL      ; Store in RAM
        INC     DI              ; Bump pointer
        IN      AL,DX           ; Read DAC blue
        MOV     CS:[DI],AL      ; Store in RAM
        INC     DI              ; Bump pointer
        LOOP    READ_DAC        ; Continue
; At this point the caller's DAC is stored in CS:WORK_DAC
; Set counter for 64 fade iterations
        MOV     BX,64           ; BX holds fade iterations
        LEA     SI,CS:WORK_DAC  ; Pointer to DAC table
;***************************|
;   read table and set DAC  |
;***************************|
; Code reads the current setting of the DAC register from the
; working storage table. This value is used to reset the DAC
; register to a darker shade
; First select DAC write operation
DARKEN_256:
        MOV     CX,256          ; Counter for 256 DAC registers
        CALL    TIME_VR_B3      ; Time with vertical retrace
        MOV     DX,03C8H        ; Pallet Address (Write mode)
        MOV     AL,0            ; Value to write
        OUT     DX,AL           ; To DAC Palette Address register
        INC     DX              ; 3C9H is DAC data register
        PUSH    SI              ; Save table pointer
;
DARKEN_DAC:
        MOV     AL,CS:[SI]      ; Load table value
        CMP     AL,0            ; Is register already black?
        JZ      DARKEN_RED      ; Go if already black
        DEC     AL              ; Darken one unit if not black
        MOV     CS:[SI],AL      ; Store value
DARKEN_RED:
        OUT     DX,AL           ; Write DAC red
        INC     SI              ; Bump pointer
;
        MOV     AL,CS:[SI]      ; Load table value
        CMP     AL,0            ; Is register already black?
        JZ      DARKEN_GREEN    ; Go if already black
        DEC     AL              ; Darken one unit if not black
        MOV     CS:[SI],AL      ; Store value
DARKEN_GREEN:
        OUT     DX,AL           ; Write DAC red
```

## 12.3.3 Color-Shift Fade-Out

One of the most common uses of color-shift animation is in producing fades. The fade-out operation takes place when the screen image gradually disappears. In the VGA system the fade-out usually consists of darkening each of the DAC registers until the black attribute is reached. This operations is called *fade-to-black*. Programs that use a fade-out may need to previously store the current DAC register setting so that the palette can be restored.

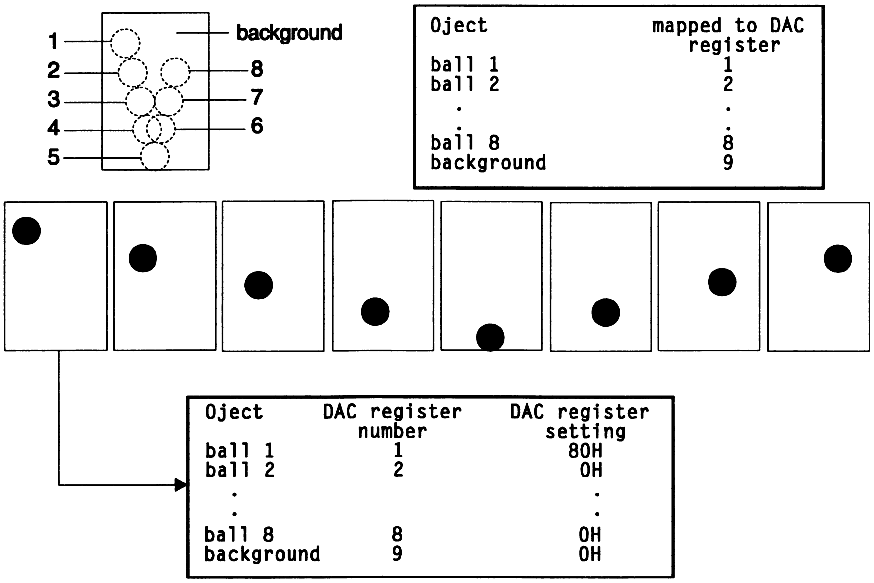
In the following procedure the fade-out can be slowed down by introducing a delay between each of the 64 incremental steps. The timing routine (called MILLI_TIME) was listed in Section 7.7.2.

```
;****************************************************************
;                    image fade-out procedure
;****************************************************************
; Code segment storage for control variables and for working
; values in DAC color table
TIME_DELAY      DW      0
WORK_DAC        DB      770 DUP (0H)
;
FADE_OUT        PROC    NEAR
; Gradually change all DAC registers from current setting to
; black to produce a fading-out of the displayed image
; On entry:
;         AX = milliseconds of time delay between palette
;              register groups
;***************************|
;     store delay value     |
;***************************|
        MOV     CS:TIME_DELAY,AX        ; To CS variable
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
;***************************|
; read and store DAC values |
;***************************|
; The procedure reads the current DAC and stores it in a working
; buffer used in the fade-out operation
        MOV     CX,256          ; Counter for 256 DAC registers
        CALL    TIME_VR_B3      ; Time with vertical retrace
        MOV     DX,03C7H        ; Pallet Address (Read mode)
        MOV     AL,0            ; Value to write
        OUT     DX,AL           ; To DAC Palette Address register
```

registers to which several screen images are mapped. When the DAC registers mapped to an image are set to the background attribute, then the object becomes invisible. When the DAC registers are set to attributes different from the background, then the image becomes visible. By manipulating the visibility of several images the object can be transformed. Figure 12.5 shows a translation transformation by color shift.

In Figure 12.5 the video buffer stores the images of eight balls, each one mapped to a different DAC register. In the initial state all eight balls are given the same attribute as the background; therefore they are all invisible. The translation transformation is performed by changing the DAC register setting for each of the ball images so that its attribute is different from that of the background, therefore making the corresponding ball visible on the screen. For example, to make ball image number 1 visible, the code changes the setting of the DAC register 1, which is mapped to this ball image. This is shown in the lower portion of Figure 12.5.

The example of color-shift animation in Figure 12.5 is the simplest one that can be contrived. In real-world applications more than one DAC register are usually mapped to each image group. In the VGA 256-color modes we could devote 16 attributes to the background and still have 48 groups of five DAC registers to map the screen objects to be animated. The possibilities of color-shift animation would be even greater in systems with color ranges in the thousands, or even the millions, as is the case in some high-resolution adapters such as those based in the TMS 340 processor. These systems are described in our book *High Resolution Video Graphics* (see Bibliography).



| Oject | mapped to DAC register |
|---|---|
| ball 1 | 1 |
| ball 2 | 2 |
| . | . |
| . | . |
| ball 8 | 8 |
| background | 9 |

| Oject | DAC register number | DAC register setting |
|---|---|---|
| ball 1 | 1 | 80H |
| ball 2 | 2 | 0H |
| . |  | . |
| . |  | . |
| ball 8 | 8 | 0H |
| background | 9 | 0H |

Figure 12.5 *Translation Transformation by Color Shift*

```
; by the caller
; On entry:
;         DS:DI -> 768-byte caller's buffer area
;**********************|
;     save caller's    |
;       context        |
;**********************|
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
;***************************|
;     read DAC registers    |
;***************************|
        MOV     CX,256           ; Counter for 256 DAC registers
        CALL    TIME_VR_B3       ; Time with vertical retrace
        MOV     DX,03C7H         ; Pallet Address (Read mode)
        MOV     AL,0             ; Value to write
        OUT     DX,AL            ; To DAC Palette Address register
        INC     DX
        INC     DX               ; 3C9H is DAC data register
READ_TO_RAM:
        IN      AL,DX            ; Read DAC red
        MOV     [DI],AL          ; Store in RAM
        INC     DI               ; Bump pointer
        IN      AL,DX            ; Read DAC green
        MOV     [DI],AL          ; Store in RAM
        INC     DI               ; Bump pointer
        IN      AL,DX            ; Read DAC blue
        MOV     [DI],AL          ; Store in RAM
        INC     DI               ; Bump pointer
        LOOP    READ_TO_RAM      ; Continue
;**********************|
;   restore caller's   |
;       context        |
;**********************|
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
DAC_TO_RAM      ENDP
```

### 12.3.2 Transformations by Color Shift

One of the most interesting applications of color-shift animation is in transforming a screen object by manipulating the visibility of several screen images. In the PC environment this effect is achieved by changing the DAC register or

```
DAC_TO_TABLE    PROC    NEAR
; Procedure to set the DAC registers to the values in a full
; color table furnished by the caller
; On entry:
;       DS:SI -> 256-entry color table, 3 colors per entry
;
        PUSH    AX                  ; Save context
        PUSH    CX
        PUSH    DX
        PUSH    SI
        CLI                         ; Interrupts off
;***************************|
;      set DAC registers    |
;***************************|
        MOV     CX,256              ; Counter for 256 DAC registers
        CALL    TIME_VR_B3          ; Time with vertical retrace
        MOV     DX,03C8H            ; Pallet Address (Write mode)
        MOV     AL,0                ; Value to write
        OUT     DX,AL               ; To DAC Palette Address register
        INC     DX                  ; 3C9H is DAC data register
SET_TO_TABLE:
        MOV     AL,[SI]             ; Load table value
        OUT     DX,AL               ; Write DAC red
        INC     SI                  ; Bump pointer
        MOV     AL,[SI]             ; Load table value
        OUT     DX,AL               ; Write DAC green
        INC     SI                  ; Bump pointer
        MOV     AL,[SI]             ; Load table value
        OUT     DX,AL               ; Write DAC blue
        INC     SI                  ; Bump pointer
        LOOP    SET_TO_TABLE        ; Continue
; Restore caller's context
        STI                         ; Interrupts ON
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
DAC_TO_TABLE    ENDP
```

### Storing the DAC Register Settings

Another operation often required by graphics software is to read and store the contents of the entire DAC register set. The following procedure performs this function:

```
;****************************************************************
;          store DAC registers in a caller's buffer
;****************************************************************
DAC_TO_RAM      PROC    NEAR
; Store current values of 768 DAC registers in buffer designated
```

DAC table of all-black codes to the corresponding BIOS service. A more compact and efficient method is the following procedure:

```
;****************************************************************
;                   all DAC registers to black
;****************************************************************
DAC_TO_BLACK    PROC    NEAR
; Procedure to write black attribute in all 768 DAC registers
; of the VGA system
;
        PUSH    AX              ; Save context
        PUSH    BX
        PUSH    CX
        PUSH    DX
        CLI                     ; Interrupts off
;**************************|
;    set DAC to black      |
;**************************|
        MOV     CX,256          ; Counter for 256 DAC registers
        CALL    TIME_VR_B3      ; Time with vertical retrace
        MOV     DX,03C8H        ; Pallet Address (Write mode)
        MOV     AL,0            ; Value to write
        OUT     DX,AL           ; To DAC Palette Address register
        INC     DX              ; 3C9H is DAC data register
SET_DAC_BLK:
        OUT     DX,AL           ; Write DAC red
        JMP     SHORT $+2       ; I/O delay
        OUT     DX,AL           ; Write DAC green
        JMP     SHORT $+2       ; I/O delay
        OUT     DX,AL           ; Write DAC blue
        LOOP    SET_DAC_BLK     ; Continue
; Restore caller's context
        STI                     ; Interrupts ON
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
DAC_TO_BLACK    ENDP
```

## Setting the DAC Registers

In Chapter 6 we listed several procedures to manipulate the DAC registers by means of BIOS services. The following procedure sets the DAC registers to the values in a table furnished by the caller. The procedure operates on all 256 DAC triplets (768 registers), all of which must be coded in the table.

```
;****************************************************************
;         set DAC to color table furnished by caller
;****************************************************************
```

```
; controller by testing bit 3 of the Input Status register 1
;
; Save caller's context
        PUSH    AX
        PUSH    DX
        MOV     DX,3DAH             ; Input Status register 1
                                    ; address in VGA mode X
VR3_CLEAR:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JNZ     VR3_CLEAR           ; Wait until bit clear
                                    ; Vertical retrace ended
VR3_START:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JZ      VR3_START           ; Wait until bit set
                                    ; Vertical retrace has started
        POP     DX                  ; Restore caller's context
        POP     AX
        RET
TIME_VR_B3      ENDP
;****************************************************************
;
TIME_VR_B0      PROC        NEAR
; Test for start of the vertical retrace cycle of the CRT
; controller by testing bit 0 of the Input Status register 1
;
; Save caller's context
        PUSH    AX
        PUSH    DX
        MOV     DX,3DAH             ; Input Status register 1
                                    ; address in VGA mode X
VR1_CLEAR:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00000001B        ; Is bit 0 set?
        JNZ     VR1_CLEAR           ; Wait until bit clear
                                    ; Vertical retrace ended
VR1_START:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00000001B        ; Is bit 0 set?
        JZ      VR1_START           ; Wait until bit clear
                                    ; Vertical retrace ended
        POP     DX                  ; Restore caller's context
        POP     AX
        RET
TIME_VR_B0      ENDP
```

## Set All DAC Register to Black

It is often necessary to set all DAC registers to black prior to a fade-in or other color-shift animation operation. This operation can be performed by passing a become invisible. A program can use this feature to produce a gradual fade-in of the video image by first setting all DAC registers to black, drawing the image while all objects are invisible, then progressively lightening each of the DAC registers until the desired final palette is obtained. The result is comparable to the fade-in operation that is commonly used in cartoon and motion picture technology. The fade-out operation follows the reverse process, that is, each of the DAC registers is diminished in brightness until the displayed image completely disappears from the screen.

## 12.3.1 VGA DAC Primitives

In order to perform color-shift animation it is convenient to have at hand a set of high-performance primitives for manipulating the DAC registers. The BIOS does contain services to set the DAC registers to the values in a table furnished by the caller and to read the DAC register contents. Some of these services are used in the routines developed in Chapter 6. Nevertheless, DAC programming is simple and straightforward, and code that accesses the VGA hardware directly has better performance and control.

### Timing Considerations

The IBM documentation for VGA systems mentions that programming operations that modify the DAC registers should be synchronized with the vertical retrace cycle of the CRT controller in order to avoid screen garbage. The VGA hardware provides three different methods for determining if the vertical retrace cycle is in progress. In Figure 5.6, we see that bit 7 of Input Status register 0 can be used to determine if a vertical retrace interrupt is in progress. However, this bit refers to the interrupt action, not the vertical retrace itself. Therefore, if the vertical retrace interrupt is not available, as is the case in many IBM and non-IBM VGA systems, or if it is not enabled, then this bit is unreliable. This is probably the reason why IBM manuals recommend that vertical retrace timing be based on either bit 3 or bit 0 of Input Status register 1, also shown in Figure 5.6.

Which of these two bits is used in vertical retrace timing depends on the type of DAC operation. For example, if timing synchronization is necessary prior to accessing the DAC data registers, then software must use bit 0 of Input Status register 1. However, if timing synchronization is necessary before accessing the DAC Read or Write Select registers, then software must use bit 3 of this register. Consequently, the programmer must have available two timing routines: one that uses the status of bit 3 and one that uses the status of bit 0 of the VGA Input Status register 1. The procedures listing follows:

```
;****************************************************************
;              vertical retrace timing procedures
;****************************************************************
TIME_VR_B3      PROC        NEAR
; Test for start of the vertical retrace cycle of the CRT
```

```
        ADD     DX,05H          ; Interrupt Status register
        IN      AL,DX           ; Read status
        OR      AL,00000001B    ; Set bit 0, preserve others
        OUT     DX,AL           ; Reset start of blanking
;*********************|
;   restore context   |
;*********************|
; Registers used by the service routine are restored from the
; stack
        POP     ES
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STI                     ; Reenable interrupts
        IRET
;****************************************************************
;                           code segment data
;****************************************************************
OLD_VECTOR_0A   DD      0       ; Pointer to original INT 0AH
                                ; interrupt
XGA_BASE        DW      0       ; Address of CRT controller
        .
        .
        .
```

The comparatively high performance of the XGA system makes possible the smooth animation of images much larger and elaborate than those that can be animated in VGA. Whenever possible the animation routine should use direct coprocessor programming (see Chapter 7) in order to minimize execution time. The system memory to video RAM pixBlt operation discussed in Section 7.4.3 can often be used in XGA animation.

## 12.3 Color-Shift Animation

In Chapter 6 we discussed several manipulations of the VGA color hardware that have possible applications to animation software. The use of the video system's color output to produce animated effects is related to the color richness of the device or mode. In this manner, in VGA mode 18, with 16 displayed colors, the options are much more limited than in VGA mode 19, or mode X, with 256 displayed colors. For this reason the discussion that follows refers to the 256-color VGA palette in modes 19 and X.

A crafty programmer can use color hardware manipulations to produce animated effects that are among the most pleasant that can be obtained in the VGA and XGA. Color-shift animation is based on the fact that the color attribute can be used to hide, display, darken, or brighten a screen object. For instance, if all the DAC registers are programmed to the same color attribute, then the entire screen appears in this color and all objects stored in the video buffer

```
        CLI                     ; Interrupts off
; Save registers
        PUSH    AX              ; Save context at interrupt time
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    ES
;**********************|
;   test for screen    |
;  blanking interrupt  |
;**********************|
; Since several hardware interrupts can be located at IRQ2 the
; software must make sure that it was screen blanking that
; originated this action. This can be done by testing bit 0 of
; the XGA Interrupt Status register
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,05H          ; Interrupt Status register
        IN      AL,DX           ; Read register contents
        TEST    AL,00000001B    ; Test start of blanking bit
        JNZ     BLK_CAUSE       ; Go if bit set
;**********************|
; chain to next handler|
;   if not blanking    |
;**********************|
; At this point the interrupt was not due to an XGA screen
; blanking interrupt. Execution is returned to the IRQ2 handler
        POP     ES              ; Restore context
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_0A
;**********************|
; animation operations |
;**********************|
BLK_CAUSE:
; At this point the handler contains the graphics operations
; necessary to perform the animation function
        .
        .
        .
;**********************|
; service routine exit |
;**********************|
; Enable 8259 interrupt controller to receive other interrupts
        MOV     AL,20H          ; Port address
        OUT     20H,AL          ; Send EOI code
; The handler must reset bit 0 of the XGA Interrupt Status
; register to clear the interrupt condition
        MOV     DX,CS:XGA_BASE  ; Display controller base address
```

```
; At this point the XGA start of blanking interrupt is active
; Program code to follow
        .
        .
        .
;****************************************************************
;                        exit routine
;****************************************************************
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires disabling the XGA screen blanking interrupt and
; restoring the original INT 0AH handler in the vector table
;**********************|
;  disable XGA screen  |
;  blanking interrupt  |
;**********************|
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,04H          ; Interrupt Enable register
        IN      AL,DX           ; Read register contents
        AND     AL,11111110B    ; Make sure bit 0 is clear
        OUT     DX,AL           ; Back to Interrupt Enable
                                ; register
;**********************|
;   restore original   |
;   INT 0AH handler    |
;**********************|
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX --> offset
        MOV     AX,CS:[SI+2]    ; AX --> segment
        MOV     DS,AX           ; segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,0AH          ; IRQ2
        INT     21H
; At this point the exiting program usually resets the video
; hardware to a text mode and returns control to the operating
; system
        .
        .
        .
;****************************************************************
;               XGA screen blanking interrupt handler
;****************************************************************
; The following routine gains control with every vertical retrace
; interrupt (approximately 70 times per second)
; The code can now perform limited video buffer update operations
; without interference
; In order to avoid interrupt reentrancy, the screen blanking
; interrupt is not reenabled until the routine has concluded
;****************************************************************
XGA_0A_INT:
```

```
; Uses DOS service 53 of INT 21H to store the address of the
; original INT 0AH handler in a code segment variable
        MOV     AH,53           ; Service request number
        MOV     AL,0AH          ; Code of vector desired
        INT     21H
; ES -> Segment address of installed interrupt handler
; BX -> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
        MOV     CS:[SI],BX      ; Save offset of original handler
        MOV     CS:[SI+2],ES    ; and segment
;**********************|
; install this INT 0AH |
;       handler        |
;**********************|
; Uses DOS service 37 of INT 21H to install the present handler
; in the vector table
        MOV     AH,37           ; Service request number
        MOV     AL,0AH          ; Interrupt code
        PUSH    DS              ; Save data segment
        PUSH    CS
        POP     DS              ; Set DS to CS for DOS service
        MOV     DX,OFFSET CS:XGA_0A_INT
        INT     21H
        POP     DS              ; Restore local data
;**********************|
;    enable IRQ2       |
;**********************|
; Clear bit 2 of the 8259 Mask register to enable the IRQ2 line
        CLI                     ; Make sure interrupts are off
        MOV     DX,21H          ; Port address of 8259 Mask
                                ; register
        IN      AL,DX           ; Read byte at port
        AND     AL,11111011B    ; Mask for bit 2
        OUT     DX,AL           ; Back to 8259 port
;**********************|
;  activate XGA screen |
;  blanking interrupt  |
;**********************|
; Reset all interrupts in the Status register
        MOV     DX,CS:XGA_BASE  ; Base address of XGA video
        ADD     DX,05H          ; Interrupt Status register
        MOV     AL,0C7H         ; All ones
        OUT     DX,AL           ; Reset all bits
; Enable the start of blanking cycle interrupt source (bit 0)
        MOV     DX,CS:XGA_BASE  ; XGA base address
        ADD     DX,04H          ; Interrupt Enable register
        IN      AL,DX           ; Read register contents
        OR      AL,00000001B    ; Make sure bit 0 is set
        OUT     DX,AL           ; Back to Interrupt Enable
                                ; register
        STI                     ; Interrupts ON
```

The XGA Interrupt Status register is also used to clear an interrupt condition. This operation is performed by the handler in order to reset the interrupt origin. The following template contains the program elements necessary for the installation and operation of a vertical retrace intercept in an XGA system:

```
;****************************************************************
;           screen blanking interrupt pulse generator
;                      for XGA systems
;****************************************************************
;
; Operations performed during installation:
; 1. The XGA port base address is stored in a code segment
;    variable named XGA_BASE
; 2. The address of the interrupt 0AH handler is saved in a
;    far pointer variable named OLD_VECTOR_0A
; 3. A new handler for interrupt 0AH is installed at the label
;    XGA_0A_INT.
; 4. The IRQ2 bit is enabled in the 8259 (or equivalent)
;    Interrupt Controller Mask register
; 5. The XGA screen blanking interrupt is enabled
;
; Operation:
; 3. The new interrupt handler at INT 0AH gains control with
;    every vertical retrace cycle of the CRT controller.
;    The software can perform limited buffer update operations
;    at this time without causing video interference
;
;****************************************************************
;                    installation routine for
;                the XGA screen blanking interrupt
;****************************************************************
; The following code enables the screen blanking interrupt on
; a XGA system and intercepts INT 0AH (IRQ2 vector)
;*********************|
;      init XGA       |
;*********************|
; XGA initialization is performed by means of the procedures
; listed in Chapter 9
        CALL    OPEN_AI; Open Adapter Interface for use
        CALL    INIT_XGA      ; Initialize XGA hardware
; The INIT_XGA procedure returns the address of the XGA register
; base in the BX register. The code stores this value in a code
; segment variable named XGA_BASE
        MOV     CS:XGA_BASE,BX ; Store in code segment variable
        MOV     AL,2           ; Select mode XGA mode number 2
                               ; 1024-by-768 pixels in 256 colors
        CALL    XGA_MODE       ; Mode setting procedure
;*********************|
;   save old INT 0AH  |
;*********************|
```

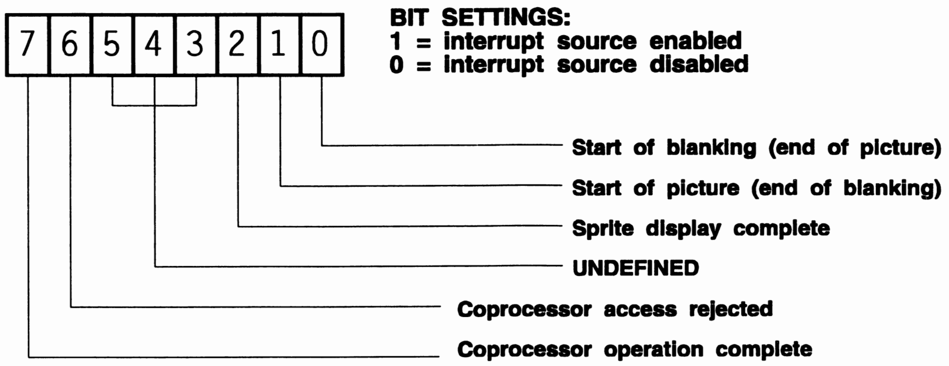
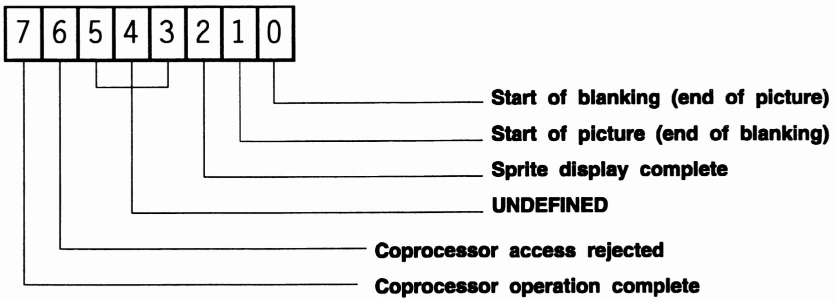

7 6 5 4 3 2 1 0
BIT SETTINGS:
1 = interrupt source enabled
0 = interrupt source disabled
Start of blanking (end of picture)
Start of picture (end of blanking)
Sprite display complete
UNDEFINED
Coprocessor access rejected
Coprocessor operation complete

Figure 12.3 *XGA Interrupt Enable Register*

### 12.2.6 XGA Screen Blanking Interrupt

The XGA documentation refers to the vertical retrace cycle as the screen blanking period. Two interrupts sources are related to the blanking period: the start of picture interrupt and the start of blanking interrupt. The start of picture coincides with the end of the blanking period. Both interrupts are enabled in the XGA Interrupt Enable register (offset 21x4H). Figure 12.3 shows a bitmap of the XGA Interrupt Enable register.

Like the VGA interrupts, the XGA video interrupts are vectored to the IRQ2 line of the 8259/A (or compatible) interrupt controller chip, which is mapped to the 0AH vector. By testing the bits in the Interrupt Status register (at offset 21x5H) an XGA program can determine the cause of an interrupt on this line. Figure 12.4 shows a bitmap of the XGA Interrupt Status register.


7 6 5 4 3 2 1 0
Start of blanking (end of picture)
Start of picture (end of blanking)
Sprite display complete
UNDEFINED
Coprocessor access rejected
Coprocessor operation complete

**BIT SETTING INTERPRETATION**

| ON READ: | ON WRITE: |
|---|---|
| 1 = interrupt occurred | 1 = clear interrupt condition |
| 0 = interrupt did not occur | 0 = no effect |

Figure 12.4 *XGA Interrupt Status Register*

```
;**********************|
; service routine exit |
;**********************|
; Enable 8259 interrupt controller to receive other interrupts
        MOV     AL,20H          ; Port address
        OUT     20H,AL          ; Send EOI code
; Reenable vertical retrace interrupt by clearing bits 4 and 5
; of the Vertical Retrace End register and then setting bit 5
; so that the interrupt is not held active
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in VRE register
        AND     AH,11001111B    ; Clear bits 4 and 5
                                ; 4 = clear vertical interrupt
                                ; 5 = enable vertical retrace
        OUT     DX,AX           ; To port
        OR      AH,00010000B    ; Set bit 4 to reset flip-flop
        OUT     DX,AX           ; To port
;
;**********************|
;   restore context    |
;**********************|
; Registers used by the service routine are restored from the
; stack
        POP     ES
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STI                     ; Reenable interrupts
        IRET
;
;****************************************************************
;                          code segment data
;****************************************************************
OLD_VECTOR_0A   DD      0       ; Pointer to original INT 0AH
                                ; interrupt
CRT_PORT        DW      0       ; Address of CRT controller
OLD_VRE         DB      0       ; Original contents of VRE
                                ; register
        .
        .
        .
```

Applications can extend the screen update time by locating the animated image as close as possible to the bottom of the video screen. In this manner the interference-free period includes not only the time lapse during which the beam is being diagonally re-aimed, but also the period during which the screen lines above the image are being scanned.

```
; interrupt (approximately 70 times per second)
; The code can now perform limited video buffer update operations
; without interference
; The vertical retrace interrupt is not reenabled until the
; routine has concluded to avoid reentrancy
;****************************************************************
;
HEX0A_INT:
        CLI                     ; Interrupts off
; Save registers
        PUSH    AX              ; Save context at interrupt time
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    ES
;
;**********************|
;  test for vertical   |
;  retrace interrupt   |
;**********************|
; Since several hardware interrupts can be located at IRQ2 the
; software must make sure that it was the vertical retrace that
; originated this action. This is done by testing bit 7 of the
; Input Status register 0, which is set if a vertical retrace
; interrupt has occurred
        MOV     DX,3C2H         ; Input Status register 0
        IN      AL,DX           ; Read byte at port
        TEST    AL,10000000B    ; Is bit 7 set?
        JNE     VRI_CAUSE       ; Go if vertical retrace
;
;**********************|
; chain to next handler|
;**********************|
; At this point the interrupt was not due to a vertical retrace
; Execution is returned to the IRQ2 handler
        POP     ES              ; Restore context
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_0A
;**********************|
; animation operations |
;**********************|
VRI_CAUSE:
; At this point the handler contains the graphics operations
; necessary to perform the animation function
        .
        .
        .
        STI                     ; Enable interrupts
; At this point the vertical retrace interrupt is active
; Program code to follow
        .
        .
        .
;
;****************************************************************
;                      exit routine
;****************************************************************
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires disabling the vertical retrace interrupt and restoring
; the original INT 0AH handler in the vector table
;
;**********************|
;   disable vertical   |
;      interrupts      |
;**********************|
; Code assumes that on program entry the vertical retrace
; was disabled
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in Vertical
                                ; Retrace End register
        OUT     DX,AX           ; To port
;
;**********************|
;   restore original   |
;   INT 0AH handler    |
;**********************|
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX -> offset
        MOV     AX,CS:[SI+2]    ; AX -> segment
        MOV     DS,AX           ; segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,0AH          ; IRQ2
        INT     21H
; At this point the exiting program usually resets the video
; hardware to a text mode and returns control to the operating
; system
        .
        .
        .
;
;****************************************************************
;             VGA vertical retrace interrupt handler
;****************************************************************
; The following routine gains control with every vertical retrace
```

```
        MOV     AH,53           ; Service request number
        MOV     AL,0AH          ; Code of vector desired
        INT     21H
; ES -> Segment address of installed interrupt handler
; BX -> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_0A
        MOV     CS:[SI],BX      ; Save offset of original handler
        MOV     CS:[SI+2],ES    ; and segment
;
;**********************|
; install this INT 0AH |
;        handler       |
;**********************|
; Uses DOS service 37 of INT 21H to install the present handler
; in the vector table
        MOV     AH,37           ; Service request number
        MOV     AL,0AH          ; Interrupt code
        PUSH    DS              ; Save data segment
        PUSH    CS
        POP     DS              ; Set DS to CS for DOS service
        MOV     DX,OFFSET CS:HEX0A_INT
        INT     21H
        POP     DS              ; Restore local data
;
;**********************|
;    enable IRQ2       |
;**********************|
; Clear bit 2 of the 8259 Mask register to enable the IRQ2 line
        CLI                     ; Make sure interrupts are off
        MOV     DX,21H          ; Port address of 8259 Mask
                                ; register
        IN      AL,DX           ; Read byte at port
        AND     AL,11111011B    ; Mask for bit 2
        OUT     DX,AL           ; Back to 8259 port
;
;**********************|
;  activate vertical   |
;  retrace interrupt   |
;**********************|
        MOV     DX,CS:CRT_PORT  ; Recover CRT base address
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        MOV     AH,CS:OLD_VRE   ; Default value in Vertical
                                ; Retrace End register
        AND     AH,11001111B    ; Clear bits 4 and 5 in VRE
                                ; Bit 4 = clear vertical
                                ; interrupt
                                ; Bit 5 = enable vertical retrace
        OUT     DX,AX           ; To port
        OR      AH,00010000B    ; Mask to set bit 4 to reenable
        OUT     DX,AX
```

```
; 1. The VGA port base address is stored in a code segment
;    variable named CRT_PORT and the default contents of the
;    Vertical Retrace End register are stored in a variable
;    named OLD_VRE
; 2. The address of the interrupt 0AH handler is saved in a
;    far pointer variable named OLD_VECTOR_0A
; 3. A new handler for interrupt 0AH is installed at the label
;    HEX0A_INT.
; 4. The IRQ2 bit is enabled in the 8259 (or equivalent)
;    interrupt controller mask register
; 5. The vertical retrace interrupt is activated
;
; Operation:
;    The new interrupt handler at INT 0AH gains control with
;    every vertical retrace cycle of the CRT controller.
;    The software can perform limited buffer update operations
;    at this time without causing video interference
;
;****************************************************************
;                    Installation routine for
;                  the vertical retrace interrupt
;****************************************************************
;
; The following code enables the vertical retrace interrupt on
; a VGA system and intercepts INT 0AH (IRQ2 vector)
;
;**********************|
;    save parameters   |
;**********************|
; System port address is saved in CS variables
        CLI                     ; Interrupts off
        MOV     AX,0H           ; Clear AX
        MOV     ES,AX           ; and ES
        MOV     DX,ES:[0463H]   ; Get CRT controller base address
                                ; from BIOS data area
        MOV     CS:CRT_PORT,DX  ; Save address in memory variable
        MOV     AL,11H          ; Offset of Vertical Retrace End
                                ; register in the CRTC
        OUT     DX,AL           ; Select this register
; Value stored in port's data register is saved in a code segment
; variable for later use by the software
        INC     DX              ; Point to Data register
        IN      AL,DX           ; Read default value in register
        JMP     SHORT $+2       ; I/O delay
        MOV     CS:OLD_VRE,AL   ; Save value in variable
;
;**********************|
;   save old INT 0AH   |
;**********************|
; Uses DOS service 53 of INT 21H to store the address of the
; original INT 0AH handler in a code segment variable
```

## 12.2.5 VGA Vertical Retrace Interrupt

Some VGA systems support an interrupt that occurs on the vertical retrace cycle while others do not. In systems that support the vertical retrace interrupt this is the most satisfactory method for obtaining a timed pulse. Its use requires programming the VGA CRT controller to generate an interrupt at the start of the vertical retrace cycle. The screen refresh rate is approximately 70 cycles per second, which is more than sufficient to achieve smooth transformations. The most important objection to this method is that it leaves very little time in which to perform image or data processing operations between timed pulses.

The programmer should keep in mind that not all PC video systems support a vertical retrace interrupt. For example, the IBM VGA Adapter is not documented to support the vertical retrace interrupt. The same applies to many VGA cards by third-party vendors. Therefore VGA programs that use the vertical retrace interrupt may not be portable to these systems.

One advantage of using the vertical retrace interrupt as a time-pulse generator is that since screen updates take place while the video system is turned off, interference is automatically avoided. The typical method of operation is to synchronize the screen update with the beginning of the vertical retrace cycle of the CRT controller. How much processing can be done while the CRT is off depends on the system hardware. In VGA this depends mainly on the type and speed of the CPU and the memory access facilities. XGA systems have their own graphics coprocessor, and for this reason, can execute considerably more processing during the vertical retrace cycle. In IBM XGA documentation the vertical retrace cycle is called the screen blanking period.

In VGA systems the smooth animation of relatively small screen objects can be executed satisfactorily by vertical retrace synchronization. As the screen objects get larger it is more difficult to update the video buffer in the short time lapse of the vertical retrace cycle. Since so many performance factors enter into the equation, it is practically impossible to give exact limits or guidelines for satisfactory animation.

It is often possible to program around the limitations of vertical retrace timing. In the first place, the image update operation can be split into two or more vertical retrace cycles. This is possible because the jerkiness frequency of 20 cycles per second is considerably less than the typical vertical retrace pulse of 70 cycles per second. However, splitting the update operations introduces programming complications, as well as an additional overhead in keeping track of which portion of the image is to be updated in each cycle.

The following coding template contains the program elements necessary for the installation and operation of a vertical retrace intercept in a VGA system:

```
;*****************************************************************
;          vertical retrace interrupt pulse generator
;                       for VGA systems
;*****************************************************************
;
; Operations performed during installation:
```

### 12.2.4 Retrace Cycle Timing

The second method for reducing interference is to synchronize the video buffer update with the one of the retrace cycles of the CRT controller. In Section 12.0.2 we mentioned that the video function is inactive during the vertical and horizontal retrace periods of the video scan (see Figure 12.1). Since the duration of the vertical retrace period is longer than the horizontal retrace, it is usually preferred in synchronization operations.

The VGA hardware provides three different methods for determining the start of the vertical retrace cycle. Bit 7 of Input Status register 0 can be used to determine if a vertical retrace interrupt is in progress. Notice that this bit refers to the interrupt action, not the vertical retrace cycle itself. PC manuals recommend that vertical retrace timing be based on either bit 3 or bit 0 of Input Status register 1, shown in Figure 5.6. Which of these two bits is used in vertical retrace timing depends on the type of synchronization necessary. A general-purpose synchronization procedure can be based on bit 3 of the Input Status register 1.

```
;****************************************************************
;  procedure to determine the start of the vertical retrace
;****************************************************************
;
TIME_VRC        PROC        NEAR
; Test for start of the vertical retrace cycle of the CRT
; controller by testing bit 3 of the Input Status register 1
;
; Save caller's context
        PUSH    AX
        PUSH    DX
        MOV     DX,3DAH             ; Input Status register 1
                                    ; address in VGA mode X
VR3_CLEAR:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JNZ     VR3_CLEAR           ; Wait until bit clear
                                    ; Vertical retrace ended
VR3_START:
        IN      AL,DX               ; Read byte at port
        TEST    AL,00001000B        ; Is bit 3 set?
        JZ      VR3_START           ; Wait until bit set
                                    ; Vertical retrace has started
        POP     DX                  ; Restore caller's context
        POP     AX
        RET
TIME_VRC        ENDP
```

The problem of vertical retrace timing resurfaces in relation to DAC programming later in this chapter.

### 12.2.3 Turning the Video Function On and Off

PC software that uses the system timer to produce a pulse for animation routines encounters interference problems. At least two methods are available to avoid or minimize display interference: turn off the CRT while the buffer is being changed, or time the buffer updates with one of the retrace cycles of the CRT controller, usually the vertical retrace. Neither method is a panacea; it is not always possible to produce smooth real time animation in the PC, particularly in VGA video systems. Applications can try either or both methods and select the better option. The following code fragment shows the processing necessary to turn off the VGA video display system:

```
;***************************|
;       turn off video      |
;***************************|
;
; Screen is turned off by setting the Clocking Mode register bit
; number 5 of the VGA Sequencer group
        MOV     DX,03C4H         ; Sequencer group
        MOV     AL,01H           ; Clocking Mode register
        OUT     DX,AL            ; Select this register
        INC     DX               ; To data port 3C5H
        IN      AL,DX            ; Read Clocking Mode register
        OR      AL,00100000B     ; Set bit 5, preserve others
        OUT     DX,AL            ; Write back to port
; At this point the VGA video display function is OFF
        .
        .
        .
```

The reverse process is necessary to turn on the VGA video display system.

```
;***************************|
;       turn on video       |
;***************************|
; Screen is turned on by clearing the Clocking Mode register bit
; number 5 of the VGA Sequencer group
        MOV     DX,03C4H         ; Sequencer group
        MOV     AL,01H           ; Clocking Mode register
        OUT     DX,AL            ; Select this register
        JMP     SHORT $ + 2      ; I/O delay
        INC     DX               ; To data port 3C5H
        IN      AL,DX            ; Read Clocking Mode register
        AND     AL,11011111B     ; Clear bit 5, preserve others
        OUT     DX,AL            ; Write back to port
; At this point the VGA video display function is ON
        .
        .
        .
```

```
;****************************************************************
HEX08_INT:
        STI                     ; Interrupts on
        PUSH    AX              ; Save registers used by routine
        PUSH    BX
        PUSH    CX              ; Other registers can be pushed
        PUSH    DX              ; if necessary
        PUSH    DS
; User video image update routine is coded at this point
        .
        .
        .
; The intercept routine maintains a code segment variable named
; TIMER_COUNT which stores a system timer pulse count. This
; variable is used to return control to the system timer
; interrupt every third timer beat, thus maintaining the
; original rate of 18.2 beats per second
        DEC     CS:TIMER_COUNT
        JZ      TIME_OF_DAY     ; Exit through time_of_day
;********************|
;    direct exit      |
;********************|
        MOV     AL,20H          ; Send end-of-interrupt code
        OUT     20H,AL          ; to 8259 interrupt controller
        POP     DS              ; Restore registers
        POP     DX
        POP     BX
        POP     AX
        IRET                    ; Return from interrupt
;********************|
;   pass to original  |
;   INT 08H handler   |
;********************|
TIME_OF_DAY:
        MOV     CS:TIMER_COUNT,2    ; Reset counter variable
        POP     DS
        POP     DX
        POP     BX
        POP     AX
        STC                     ; Continue processing
        JMP     DWORD PTR CS:OLD_VECTOR_08
        IRET
;********************|
;  code segment data  |
;********************|
TIMER_COUNT     DB      2       ; Timer counter
OLD_VECTOR_08   DD      0       ; Far pointer to original INT 08H
        .
        .
        .
CODE    ENDS
```

```
        .
        .
        .
;*************************|
;       exit routine      |
;*************************|
; Before the program returns control to the operating system
; it must restore the hardware to its original state. This
; requires resetting the time speed to 18.2 beats per second
; and reinstalling the BIOS interrupt handler in the vector
; table
;**********************|
;  reset system timer  |
;**********************|
; Original divisor is 65536
        CLI                     ; Interrupts off while write
                                ; LSB then MSM
                                ; xxxx 011x binary system
        OUT     43H,AL
        MOV     BX,65535        ; Default divisor
        MOV     AL,BL
        OUT     40H,AL          ; Send LSB
        MOV     AL,BH
        OUT     40H,AL          ; Send MSB
;**********************|
;   restore INT 0AH    |
;**********************|
        PUSH    DS              ; Save program's DS
        MOV     SI,OFFSET CS:OLD_VECTOR_08
; Set DS:DX to original segment and offset of keyboard interrupt
        MOV     DX,CS:[SI]      ; DX -> offset
        MOV     AX,CS:[SI+2]    ; AX -> segment
        MOV     DS,AX           ; Segment to DS
        MOV     AH,25H          ; DOS service request
        MOV     AL,08H          ; Interrupt number
        INT     21H
        POP     DS
        STI                     ; Interrupts on again
; At this point the exiting program usually resets the video
; hardware to text mode and returns control to the operating
; system
        .
        .
        .
;****************************************************************
;                       new INT 08H handler
;****************************************************************
; The handler is designed so that a new timer tick cannot take
; place during execution. This is ensured by not sending the 8259
; end-of-interrupt code until the routine's processing is
; complete
;       2. Speed up system timer by a factor of 3
;       3. Set INT 08H vector to routine in this module
;****************************************************************
;*********************|
;   save old INT 08H  |
;*********************|
; Uses DOS service 53 of INT 21H
        MOV     AH,53               ; Service request number
        MOV     AL,08H              ; Code of vector desired
        INT     21H
; ES -> Segment address of installed interrupt handler
; BX -> Offset address of installed interrupt handler
        MOV     SI,OFFSET CS:OLD_VECTOR_08
        MOV     CS:[SI],BX          ; Save offset of original handler
        MOV     CS:[SI+2],ES        ; and segment
;*********************|
;   speed up system   |
;     timer by 3      |
;*********************|
; Original divisor is 65536
; New divisor (65536/3) = 21845
;       CLI                         ; Interrupts off while write
                                    ; LSB then MSM
                                    ; xxxx 011x binary system
        OUT     43H,AL
        MOV     BX,21845            ; New divisor
        MOV     AL,BL
        OUT     40H,AL              ; Send LSB
        MOV     AL,BH
        OUT     40H,AL              ; Send MSB
;*********************|
; set new INT 08H in  |
;     vector table    |
;*********************|
; Mask off all interrupts while changing INT 08H vector
        CLI
; Save mask in stack
        IN      AL,21H              ; Read 8259 mask register
        PUSH    AX                  ; Save in stack
        MOV     AL,0FFH             ; Mask off IRQ0 to IRQ7
        OUT     21H,AL              ; Write to 8259 mask register
; Install new interrupt vector
        MOV     AH,25H
        MOV     AL,08H              ; Interrupt code
        MOV     DX,OFFSET HEX08_INT
        INT     21H
; Restore original interrupt mask
        POP     AX                  ; Recover mask from stack
        OUT     21H,AL              ; Write to 8259 mask register
        STI                         ; Set 80x86 interrupt flag
; At this point the graphics program continues execution
```

result is that an animation pulse created by loop methods is difficult to estimate, leading to nonuniform or unpredictable movement of the animated object.

### 12.2.2 Reprogramming the System Timer

A time-pulse source available in the PC is the system's timer pulse. This pulse, which can be intercepted by an application, beats at the default rate of approximately 18.2 times per second. However, an application can reprogram the system timer to generate a faster rate. An interrupt intercept routine can be linked to the system timer so that the program receives control at every timer beat. If it were not for interference problems, the system timer intercept would be an ideal beat generator for use in animation routines.

The following code fragment installs a system timer intercept routine. The installation routine accelerates the system timer from 18.2 to 54.6 beats per second, or three times the original rate.

```
;****************************************************************
;            system timer-driven pulse generator
;                  for animation programming
;****************************************************************
;
; Changes performed during installation:
; 1. The BIOS system timer vector is stored in a code segment
;    variable
; 2. The timer hardware is made to run three times faster to ensure
;    a beat that is close to the critical flicker frequency
; 3. A new service routine for INT 08H is installed in the
;    program's address space
;
; Operation:
;    The new interrupt handler at INT 08H gains control with
;    every beat of the system timer. The program maintains a
;    beat counter in the range 0 to 2. Every third beat
;    (counter = 2) execution is passed to the original INT 08H
;    handler in the BIOS in order to preserve the timer-dependent
;    services
;
CODE    SEGMENT
START:
        .
        .
        .
;
;****************************************************************
;            installation routine for INT 08H handler
;****************************************************************
; Operations:
;       1. Obtain vector for INT 08H and store in a CS variable
;          named OLD_VECTOR_08
; Set the Graphics Controller function select field of the Data
; Rotate register to the XOR mode
        MOV     DX,03CEH    ; Graphic Controller port address
        MOV     AL,3        ; Select Data Rotate register
        OUT     DX,AL
        INC     DX          ; 03CFH register
        MOV     AL,00011000B   ; Set bits 3 and 4 for XOR
        OUT     DX,AL
```

Many conventional graphics operations, such as pixBlt and text display functions, require that the function select bits of the Data Rotate register be set for normal operation. The following code fragment shows the necessary processing:

```
; Set the Graphics Controller function select field of the Data
; Rotate register to the normal mode
        MOV     DX,03CEH    ; Graphic Controller port address
        MOV     AL,3        ; Select Data Rotate register
        OUT     DX,AL
        INC     DX          ; 03CFH register
        MOV     AL,00000000B   ; Reset bits 3 and 4 for normal
        OUT     DX,AL
```

The procedure named LOGICAL_OP listed in Section 6.2.2 can be used to set the function select field of the Graphics Controller Data Rotate register to any one of four possible logical modes.

## 12.2 Generating the Time Pulse

Time-pulse animation is a real time technique by which a screen object is successively displayed and erased at a certain rate. Ideally, the redraw rate in time-pulse animation should be higher than the critical jerkiness frequency of 20 images per second, although, in practice, the time pulse is often determined by the screen refresh rate.

### 12.2.1 Looping Techniques

The programmer has several methods of producing the timed pulse at which the animated image is updated. Which method is selected depends on the requirements of the application as well as on the characteristics of the video display hardware. The simplest method for updating the screen image of an animated object is by creating an execution loop to provide some form of timing device. But the loop must include not only the processing operations for updating the screen image but also one or more polling routines. In addition, the loop's execution can be interrupted by hardware devices requiring processor attention. Another factor that can affect the precision of the loop timing is the processor speed and memory access facilities of the particular machine. The

However, if the same XOR mask is used over a bright green background the resulting pixel is blue, as in the following example:

```
                 I R G B
background   =   1 0 1 0   (bright green)
XOR mask     =   1 0 1 1
                 -------
image        =   0 0 0 1   (blue)
```

This characteristic of XOR operations, whereby an object's color changes as it moves over different backgrounds, can be an advantage or a disadvantage in graphics applications. For example, a marker symbol conventionally displayed disappears as it moves over a background of its same color, while a marker displayed by means of a logical XOR can be designed to be visible over all possible backgrounds. On the other hand, the color of a graphics object might be such an important characteristic that any changes during display operations would be objectionable. Figure 12.2 graphically shows how the XOR operation changes the attributes of an object (circle) as it is displayed over different backgrounds.

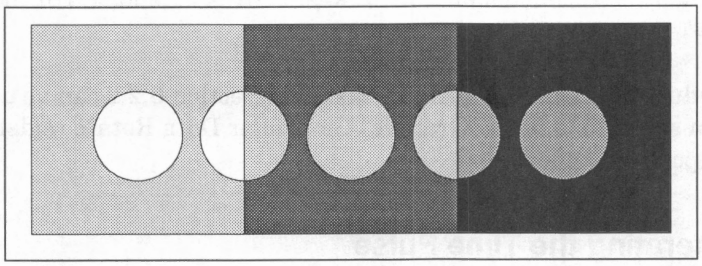


Figure 12.2 *Effect of XOR Operation*

This peculiar effect of XOR operations on the object's color may not be objectionable, and even advantageous under some conditions, but in other applications it could make this technique unsuitable. More advanced video graphics systems include hardware support for animated imagery. In XGA, for example, the sprite mechanism allows for the display and movement of marker symbols or icons independently of the background. Therefore the XGA programmer can move the sprite symbol simply by defining its new coordinates. The XGA hardware takes care of erasing the old marker and restoring the underlying image.

## 12.1.1 Programming the Function Select Bits

To make possible the XOR operation, the software must manipulate the function select bits of the Graphics Controller Data Rotate register (see Figure 5.9). The following code fragment shows the required processing:

Several techniques have been devised for performing the redraw-erase cycle required in figure animation. The most direct method is to save that portion of the screen image that is to be occupied by the object. The object can then be erased by redisplaying the saved image. The problem with this double pixBlt is that it requires a preliminary, and time-consuming, read operation to store the screen area that is to be occupied by the animated object. Therefore the redraw-erase cycle is performed by a video-to-RAM pixBlt (save screen), RAM-to-video pixBlt (display object), and another RAM-to-video pixBlt (restore screen).

A faster method of erasing and redrawing the screen is based on the properties of the *logical exclusive or* (XOR) operation. The action of the logical XOR is that a bit in the result is set if both operands contain opposite values. Consequently, XORing the same value twice restores the original contents, as in the following example:

```
           10000001B
XOR mask   10110011B
           ---------
           00110010B
 OR mask   10110011B
           ---------
           10000001B
```

Notice that the resulting bitmap (10000001B) is the same as the original one. The XOR method can be used in EGA, VGA, and SuperVGA systems because the Data Rotate register of the Graphics Controller can be programmed to write data normally, or to AND, OR, or XOR the CPU data with the one in the latches. In XGA systems, mix mode number 06H produces a logical XOR of source and destination pixels.

The logical XOR operation provides a convenient and fast way for consecutively drawing and erasing a screen object. Its main advantage is that it does not require a previous read operation to store the original screen contents. This results in a faster and simpler read-erase cycle. The XOR method is particularly useful when more than one animated object can coincide on the same screen position since it ensures that the original screen image is always restored.

The disadvantage of the XOR method is that the resulting image depends on the background. In other words, each individual pixel in the object displayed by means of a logical XOR operation is determined both by the XORed value and by the present pixel contents. For example, the following XOR operation produces a red object (in IRGB format) on a bright white screen background:

```
               I R G B
background  =  1 1 1 1   (bright white)
XOR mask    =  1 0 1 1
               -------
image       =  0 1 0 0   (red)
```

Some of the original graphics systems in IBM microcomputers were prone to a form of display interference called *snow*. The interference occurs when a video buffer update coincides with the screen refresh. CGA programmers soon discovered the unsightly effect could be avoided, or considerably reduced, by synchronizing the buffer updates with the period of time that the electron gun was turned off during the horizontal or the vertical retrace. EGA and VGA systems are designed to avoid this form of interference when conventional methods of buffer update are used. However, the interference problem reappears when an EGA, VGA, or XGA screen image has to be updated at short time intervals, as is the case in animation.

The frequent screen updates required by most animation routines must be timed with the period during which the electron gun is off in order to avoid interference. Because vertical retrace takes longer than the horizontal it is sometimes preferred for synchronization purposes. This requirement, which applies to EGA, VGA, XGA, and SuperVGA systems, imposes a substantial burden on programs that perform animated graphics. For example, the screen refresh period in VGA graphics modes takes place at an approximate rate of 70 times per second. Since the individual images must be updated in the buffer while the electron gun is off, this gives the software one-seventieth of a second to replace the old image with the new one. How much buffer update can be performed in this time period is the most limiting factor in programming smooth, real time animation on the PC.

Notice that a screen refresh rate of approximately one-seventieth of a second considerably exceeds the critical jerkiness frequency of one-twenty-fourth of a second used as the image refresh rate in motion picture technology (see Chapter 1). This difference is related to the time period required for the human eye to adjust to a light intensity change and detect flicker. We speak of a *critical flicker frequency*, as different from the critical jerkiness frequency mentioned. The motion picture projector contains a rotating diaphragm that blackens the screen only during the very short interval required to move the film to the next frame. This allows projection speeds to take place at the critical jerkiness rate rather than at the flicker rate. By the same token, a computer monitor must adjust the screen refresh cycle to this critical flicker frequency.

## 12.1 XOR Animation

In order to animate a screen object its image must be erased from the current screen position before being redrawn at the new position. In this respect animation programmers sometimes speak of a draw-erase-redraw cycle. If the object is not erased from the video display, its movement would leave an image track on the display surface. In lateral translation an object appears to move across the screen, from left to right, by progressively redrawing and erasing its screen image at consecutively larger $x$ coordinates. Notice that erasing the screen object is at least as time consuming as drawing it, since each pixel in the object must be changed to its previous state.

the frame-by-frame projection of a set of progressively changing images of an object. It is this collection of smoothly changing images that we call the animation image set. If the rate at which the individual images are shown on the video display is close to the critical rate of 22 images per second, then the animation appears smooth and pleasant. However, if the software cannot approximate this critical rate, the user perceives a disturbing flicker and the animation appears coarse and bumpy to various degrees.

The image retention phenomena imposes performance requirements on real time animated systems. If a computer animation program is to create a smooth and pleasant effect, all the manipulations and changes from image to image must be performed in less than one-twentieth of a second. For this reason, raster scan video systems with bit-mapped image buffers, such as those in IBM microcomputers, are not well suited for computer animation.

### 12.0.2 Avoiding Interference

A raster scan display system is based on scanning each horizontal row of screen pixels with an electron beam. The pixel rows are usually scanned starting at the top-left screen corner and ending at the bottom-right corner. In this context each pixel row is called a *scan line*. At the end of each scan line the electron beam is turned off while the gun is re-aimed to the start of the next line. This period is called the *horizontal retrace*. When this row-by-row process reaches the bottom scan line, the beam is turned off again while the gun is re-aimed to the top-left screen corner. The period of time required to re-aim the electron gun from the bottom-right corner of the screen to the top-left corner is known as the *vertical retrace* or *screen blanking* cycle. Figure 12.1 shows the scan and retrace cycles.

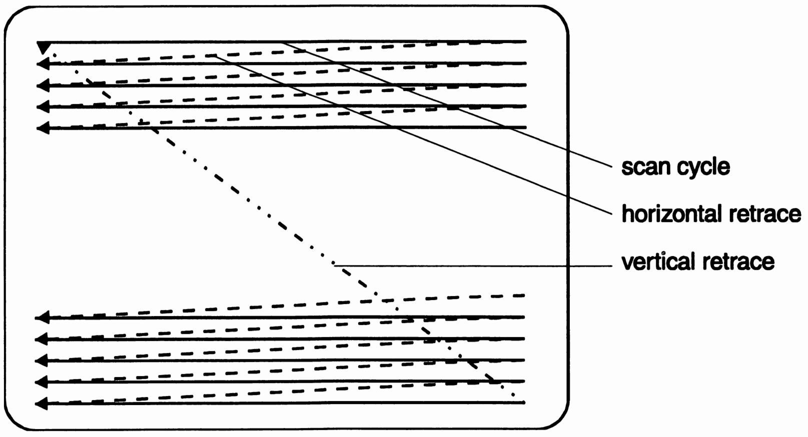


Figure 12.1 *CRT Scan and Retrace Cycles*

Chapter

# 12

# Time-Pulse and Color-Shift Techniques

## 12.0 The Animated Image Set

Video animation often depends on the display of a series of images, sometimes called the *image set*. In some forms of animation the images themselves are progressively changed to form the image set. For example, panning animation is based on changing the portion of the image that is visible on the viewport. Other geometrical transformations can be used to generate the image set. We have seen how scaling and rotation transformations are applied to a graphical object in order to simulate its approaching the viewer. In all cases, animation in real time requires two separate programming steps: the creation of an image set and the sequential display of these images.

Many graphics and nongraphics techniques are used in the creation of an image set that follows a predefined pattern of change. The image set can be generated by performing geometrical transformations on the display file. Hand-drawn or optically scanned bitmaps are also used to create the image set. Notice that the creation of the image set need not take place in real time; it is its display that is time-critical. But whether the image set is encoded in vector commands or in bitmaps, the actual animation requires displaying these images consecutively, in real time, and ideally, at a rate that is not less than the critical flicker frequency. In the following sections we discuss some programming methods used for displaying the animation image set in real time.

### 12.0.1 Visual Retention

In Chapter 1 we mentioned that the human visual organs retain, for a short time, the images of objects that no longer exist in the real world. This physiological phenomenon makes possible the creation of an illusion of animation by

In this case the software can access any character map by adding a multiple of 16 to the start address of the font table. In other words, the offset of any desired character map is the product of its ASCII code and the number of bytes in the character maps. However, in PLC printer fonts not all the character maps are of identical size. Therefore, in a typical case the character map for the letter "M" is larger than the character map for the letter "l." This complicates the calculations necessary for finding the start of a desired character map in the font table and in obtaining its specific horizontal and vertical dimensions. Procedures for using PLC fonts for display type can be found in our book *Graphics Programming Solutions* (see Bibliography).

### 11.3.3 Text Animation

We have seen that text characters are usually sets of individual bitmaps, one for each letter or symbol. Less frequently, the animation programmer deals with text characters defined in vector form, as is the case in the Postscript language and in other scalable fonts. In either case, the text characters are individual graphics entities that can be manipulated and animated as such, following the methods and operations already described.

```
        JMP     DISPLAY_BYTE
; Index to next row
NEXT_ROW1:
; Test for last graphic row
        DEC     BL                  ; Row counter
        JZ      GRAPH_END           ; Done, exit
        MOV     BH,CS:BYTES         ; Reset byte counter
        INC     DX                  ; Bump y coordinate control
        MOV     CX,CS:X_COORD       ; Reset x coordinate control
        JMP     BYTE_ENTRY
GRAPH_END:
        RET
FINE_PATTERN    ENDP
```

## Display Type

The use of character generator software and BIOS character tables, as described in the previous paragraphs, considerably expands the programmer's control over text display on the VGA graphics modes. However, the BIOS character sets consist of relatively small symbols. Many graphics applications require larger characters (sometimes called display type) for use in logos, titles, headings, or other special effects. Since the largest character sets available in BIOS are the 8-by-16 and 9-by-16 fonts, the programmer is left to his or her own resources in this matter.

There are several ways of creating, or imitating, display type screen fonts. These include the use of scalable character sets, the design of customized screen font tables, the adaptation of printer fonts to screen display, the enlargement of existing fonts, and even the artistic design of special letters and logos. Which method is suitable depends on particular needs and the availability of resources. Ideally, the display programmer would have at hand scalable text fonts in many different typefaces and styles. In fact, some sophisticated graphics programs and programming environments furnish screen versions of the Postscript language, which comes close to achieving this almost-ideal level of text display control.

## PCL Fonts as Display Type

One source of display type often available to the program designer is in the form of printer fonts in *Printer Control Language* (PCL) format. These fonts, originally developed by Hewlett-Packard for use in their LaserJet line of laser printers, are a collection of high-quality bitmaps similar to the ones in the BIOS character set previously described. The PCL font, usually stored in a disk file, can be loaded into the applications memory space in a similar manner to the BIOS character sets.

A major difference between the BIOS screen fonts and the printer fonts in PCL format is that the former have an identical pattern for all the text characters; that is, all character maps occupy the same memory space. For example, in BIOS each 8-by-16 font character map takes up 16 bytes of storage.

```
; FINE_PATTERN PROCEDURE
; Scratchpad variables for FINE_PATTERN
COUNT_8         DB      0       ; 8-bit counter
BYTES           DB      0       ; Block byte-width storage
X_COORD         DW      0
;
FINE_PATTERN    PROC    FAR
; Display a bitmap at a pixel boundary
;
; On entry:
;       CX = x coordinate to start display
;       DX = y coordinate to start display
;       BH = number of bytes per bitmap row (map's x dimension)
;       BL = number of rows in bitmap (map's y dimension)
;       DI -- start of bitmap
;       AL = I R G B color code
;
; Note: this procedure calls a virtual, device-independent
;       routine named PIXEL_WRITE, which sets an individual
;       screen pixel in the host video system. The programmer
;       must code a real pixel-set procedure for the code to
;       execute.
;
; Initialize registers
        MOV     CS:COUNT_8,8    ; Prime bit counter
        MOV     CS:BYTES,BH     ; Store byte count
        MOV     CS:X_COORD,CX   ; and x coordinate of block
DISPLAY_BYTE:
        MOV     AH,[DI]         ; Byte to be displayed
TEST_BIT:
        TEST    AH,10000000B    ; Is high bit set?
        JZ      NEXT_BIT        ; Bit not set
; Set the pixel
        PUSH    AX              ; Save entry registers
        PUSH    BX
        CALL    PIXEL_WRITE     ; (see note in procedure header)
        POP     BX              ; Restore registers
        POP     AX
NEXT_BIT:
        SAL     AH,1            ; Shift AH to test next bit
        INC     CX              ; Bump x coordinate counter
        DEC     CS:COUNT_8      ; Bit counter
        JZ      NEXT_BYTE1      ; Exit if counter rewound
        JMP     TEST_BIT        ; Continue
; Index to next byte in row, if not at end of row
NEXT_BYTE1:
        DEC     BH              ; Bytes per row counter
        JZ      NEXT_ROW1       ; End of graphic row
BYTE_ENTRY:
        INC     DI              ; Bump graphic code pointer
        MOV     CS:COUNT_8,8    ; Reset bit counter
        MOV     AX,WORD PTR [SI]    ; x axis character spacing
        MOV     CS:X_DISPL,AX       ; Store in variable
        ADD     SI,2
        MOV     AX,WORD PTR [SI]    ; y axis character spacing
        MOV     CS:Y_DISPL,AX
        ADD     SI,2
        MOV     AL,BYTE PTR [SI]    ; IRGB color code
        MOV     CS:IRGB_CODE,AL     ; Store color code
        INC     SI                  ; SI == to message text
;*************************|
;     display text line   |
;*************************|
; At this point
;       CX = x coordinate pixel address to start display
;       DX = y coordinate pixel
;       SI == first character in text message
NEXT_FINE:
        MOV     DI,CS:FINE_ADD  ; Offset of RAM font table
        SUB     AX,AX           ; Clear high byte
        MOV     AL,BYTE PTR [SI]     ; ASCII symbol
; Test for embedded control code
        CMP     AL,0            ; End of message
        JNE     NOT_FINE_END
        RET
NOT_FINE_END:
; At this point AL holds ASCII character code, which is the
; character's offset in the font
; Calculate character bitmap offset in font
        MUL     CS:MAP_X    ; Offset times number of columns
        MUL     CS:MAP_Y    ; and times number of rows
                            ; in bitmap
        ADD     DI,AX       ; DI => font to be displayed
; DI == bitmap to be displayed
        MOV     BL,CS:MAP_Y     ; Map's y dimension
        MOV     BH,CS:MAP_X     ; Map's x dimension
        MOV     AL,CS:IRGB_CODE ; Color code for display
        PUSH    CX              ; Save x and y coordinates
        PUSH    DX
        CALL    FAR PTR FINE_PATTERN
        POP     DX              ; Restore coordinates
        POP     CX
; Index to next character
        ADD     CX,CS:X_DISPL   ; Add x displacement
        ADD     DX,CS:Y_DISPL   ; and y displacement
        INC     SI              ; Bump message pointer
        JMP     NEXT_FINE
;
FINE_TEXT       ENDP
;****************************************************************
;               support procedure for FINE_TEXT
;****************************************************************
```

```
; same dimensions. Display position is defined at a pixel
; boundary. Code assumes an initialized VGA graphics system
;
; Message format:
;  OFFSET  STORAGE   CONTENTS
;    0      word     x coordinate start address (pixel number)
;    2      word     y coordinate start address (pixel number)
;    4      word     x axis pixel spacing between characters
;    6      word     y axis pixel spacing between characters
;    8      byte     I R G B color code
;    9      ----     first character in text message
;
; Font table format:
; Each font table is preceded by two bytes that determine its
; dimensions, as follows:
;   Byte at font table - 1 = number of pixel rows in bitmap
;   Byte at font table - 2 = number of horizontal bytes in bitmap
;
; Embedded codes:
;                    00H = end of message
; Note: since this is a single-line function no end of line code
; is implemented
;
; On entry:
;     DS:SI == message
;     DS:DI == RAM font table
;
; Code segment data for scratchpad variables
FINE_ADD        DW      0       ; Offset of font table in RAM
MAP_X           DB      0       ; Horizontal bytes in bitmap
MAP_Y           DB      0       ; Number of pixel rows in map
X_DISPL         DW      0       ; x character displacement
Y_DISPL         DW      0       ; y character displacement
IRGB_CODE       DB      0
;
FINE_TEXT       PROC    FAR
; Save start address of font table
        MOV     CS:FINE_ADD,DI      ; In memory variable
;*************************|
;  obtain font dimensions  |
;*************************|
; Load font dimension into registers and store in variables
        MOV     BH,[DI-2]       ; x dimension into BH (bytes)
        MOV     BL,[DI-1]       ; y dimension into BL (bits)
        MOV     CS:MAP_X,BH         ; Save map dimensions
        MOV     CS:MAP_Y,BL
; Set up pointers
        MOV     CX,WORD PTR [SI]    ; x axis pointer
        ADD     SI,2
        MOV     DX,WORD PTR [SI]    ; y axis pointer
        ADD     SI,2
        LOOP    MOVE_FONT
; At this point the designated character set is resident in the
; caller's buffer
        RET
FONT_TO_RAM     ENDP
```

In loading a BIOS character font to the RAM memory, the caller can precede the font storage area with two data bytes that encode the font's dimensions. For example, the storage area for the BIOS 8-by-8 font can be formatted as follows:

```
;*********************|
;   storage for BIOS   |
;   symmetrical font   |
;*********************|
; RAM storage for symmetrical font table from BIOS character maps
; Each font table is preceded by two bytes that determine its
; dimensions, as follows:
;   Byte at font table - 1 = number of pixel rows
;   Byte at font table - 2 = number of horizontal bytes
;
; 1 x 8 built in ROM font
                DB      1       ; Bitmap x dimension, in bytes
                DB      8       ; Bitmap y dimension, in bytes
FONT_1X8        DB      2048 DUP (00H)
```

Note that 2048 bytes are reserved for the 8-by-8 BIOS font, which contains 256 character maps of 8 bytes each ($256 \times 8 = 2048$). By the same token, the 1-by-16 character font would require 4096 bytes of storage. Once the BIOS font table is resident in the caller's memory space, it can be treated as a collection of bitmaps, one for each character in the set. In this manner the programmer is able to control the screen position of each character at the pixel level. Consequently, the spacing between characters, which is necessary in line justification, also comes under software control; as does the spacing between text lines, and even the display of text messages at screen angles.

The procedure named FINE_TEXT (listing follows) allows the display of a single text line starting at any desired pixel location and using any desired spacing between characters on the horizontal and the vertical axes. This means that if the vertical spacing byte is set to zero in the text header block, all the characters are displayed on a straight line in the horizontal plane. However, by assigning a positive or negative value to this parameter, the programmer using this procedure can display a text message skewed at a screen angle.

```
;***************************************************************
;                    character generator procedure
;***************************************************************
; Character generator for displaying text messages using a font
; table located in RAM. All bitmaps in the font must have the
```

```
;****************************************************************
;  procedure to move a VGA font table to the caller's memory
;****************************************************************
FONT_TO_RAMPROCFAR
;
; On entry:
;          AL = ROM character map desired
;                8 = 8-by-8 font
;               14 = 8-by-14 font
;               16 = 8-by-16 font
;       DS:DI -- caller's RAM buffer for font storage
;                 requires 2K for 8-by-8 font and 4K for 8-by-14
;                 and 8-by-16 font
;
;**********************|
;     select font      |
;**********************|
        PUSH    DI                  ; Save destination address
        CMP     AL,8                ; Test for 8-by-8 font
        JE      LOAD_8X8            ; Go to 8-by-8 load
        CMP     AL,14               ; Test for 8-by-14 font
        JE      LOAD_8X14           ; Go to 8-by-14 load
; Default is 8-by-16 font
        MOV     BH,6                ; BIOS entry parameter
        MOV     CX,2048             ; Word length of font table
        JMP     FONT_ADDR           ; Go to load routine
LOAD_8X8:
        MOV     BH,3                ; BIOS entry parameter
        MOV     CX,1024             ; Word length of font table
        JMP     FONT_ADDR           ; Go to load routine
LOAD_8X14:
        MOV     BH,2                ; BIOS entry parameter
        MOV     CX,2048             ; Word length of font table
;*********************|
;   get font address  |
;*********************|
FONT_ADDR:
        PUSH    CX                  ; Save word length counter
        MOV     AH,17               ; BIOS character generator
                                    ; service
        MOV     AL,48               ; Subservice for font address
        INT     10H
        POP     CX                  ; Restore counter
        POP     DI                  ; Restore address for font
; At this point ES:BP -- ROM address of character table
        MOV     SI,BP               ; Sep up pointer
MOVE_FONT:
        MOV     AX,ES:[SI]          ; Get 2 bytes from ROM
        MOV     [DI],AX             ; Store them in RAM
        ADD     SI,2                ; Bump pointers
        ADD     DI,2
```

VGA systems contain three complete character fonts and two supplemental fonts. The characteristics of these fonts are shown in Table 11.1.

Table 11.1 *BIOS Character Sets in VGA Systems*

| CHARACTER BOX SIZE | MODE |
|---|---|
| 8-by-8 | 0, 1, 2, 3, 4, 5, 13, 14, and 19 |
| 8-by-14 | 0, 1, 2, 3, 15, and 16 |
| 8-by-16 | 17 and 18 |
| * 9-by-14 | 7 |
| * 9-by-16 | 0, 1, and 7 |

note: * = supplemental set

The supplemental character sets (Table 11.1) do not contain all of the 256 character maps of the full sets, but only those character maps that are different in the 9-bit-wide fonts. In the process of loading a 9-bit character set, the BIOS first loads the corresponding 8-bit character maps and then overwrites the ones that need correction and that appear in the supplemental set. This mechanism is transparent to the programmer, who sees a full set of 9-by-14 or 9-by-16 characters.

## 11.3.2 Developing a Character Generator

PC graphics applications can perform simple character display operations by means of the BIOS functions, but for many purposes these functions are too limited. Perhaps the most obvious drawback of character display by means of BIOS services is that the text characters must conform to a grid of columns and rows determined by the selected character font and the active video mode. For example, a graphics program executing in VGA mode 18 can use BIOS service number 9, interrupt 10H, to display screen text using the 8-by-16 character font. However, in this case, the program is constrained to a text screen composed of 80 character columns by 30 rows and is not able to locate text outside this grid.

### Moving a BIOS Font to RAM

VGA graphics software can obtain considerable control in text display functions by operating its own character generator, in other words, by manipulating the text character maps as a regular bitmap. The process can often be simplified if the existing character maps are suitable to the program's purpose. In VGA systems the most easily available character maps are the BIOS character sets. (See Table 11.1.) The software can gain the necessary information regarding the location of any one of the BIOS character maps by means of service number 17, subservice number 48, of interrupt 10H. Once the address of the character table is known, the code can move all or part of this table to its own address space, where it becomes readily accessible. The following procedure shows the processing necessary for loading one of three VGA character sets into the caller's memory space:

```
        MOV     DL,BYTE PTR DS:[BP]  ; Restore start column to DL
        INC     DH                   ; Row control register
;
; Set cursor using BIOS service number 2 of interrupt 10H
        PUSH    AX                   ; Save entry registers
        PUSH    BX
        PUSH    DX
        MOV     AH,2                 ; Service request number
        MOV     BH,0                 ; Assume display page No. 0
        INT     10H
        POP     DX                   ; Restore registers
        POP     BX
        POP     AX
        JMP     CHAR_WRITE
END_TEXT:
        RET
;
TEXT_BLOCK      ENDP
```

Notice that in this code fragment, the first byte in the header encodes the screen row at which the message is to be displayed, the second byte encodes the screen column, and the third byte encodes the color code. Since the procedure operates in any text or graphics mode, the range and encodings for these parameters depend on the active mode.

## BIOS Character Sets

The BIOS stores several sets of text characters encoded in bitmap form. Figure 11.8 shows the bitmap of the letter "A" as found in one of the BIOS fonts.

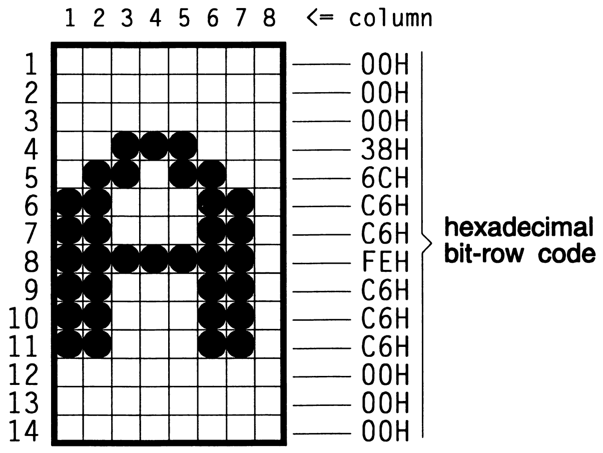


Figure 11.8 *Bitmap of BIOS Text Character*

```
; Set cursor using BIOS service number 2 of INT 10H
        PUSH    AX              ; Save entry registers
        PUSH    BX
        PUSH    DX
        MOV     AH,2            ; Service request number
        MOV     BH,0            ; Assume display page No. 0
        INT     10H
        POP     DX              ; Restore registers
        POP     BX
        POP     AX
        INC     DI              ; Bump pointer to attribute
        MOV     BL,[DI]         ; Get color code into BL
CHAR_WRITE:
        INC     DI              ; Bump to message start
        MOV     AL,[DI]         ; Get character
        CMP     AL,0FFH         ; End of line ?
        JE      BUMP_ROW        ; Next row
        CMP     AL,0            ; Test for terminator
        JZ      END_TEXT        ; Exit routine
;
;**********************|
;    display character |
;   using BIOS service |
;**********************|
; Display character in AL and using the color code in BL
        MOV     AH,9            ; BIOS service request number
        MOV     BH,0            ; Page
        MOV     CX,1            ; No repeat
        INT     10H
;
;*********************|
;     bump cursor     |
;   to next character |
;*********************|
        INC     DL
; Set cursor using BIOS service number 2 of interrupt 10H
        PUSH    AX              ; Save entry registers
        PUSH    BX
        PUSH    DX
        MOV     AH,2            ; Service request number
        MOV     BH,0            ; Assume display page No. 0
        INT     10H
        POP     DX              ; Restore registers
        POP     BX
        POP     AX
        JMP     CHAR_WRITE
;
;*********************|
;      next row       |
;*********************|
BUMP_ROW:
```

### 11.3.1 Text Display Operations in BIOS

BIOS service number 9, interrupt 10H, can be used in standard alphanumeric and graphics modes to display a character at the current cursor position. Notice that this statement refers to the BIOS standard modes, and not to the non-standard ones such as VGA mode X (see Chapter 7). Service number 9 is the only BIOS character display service that can be used in a graphics mode, while several other BIOS text services can be used in alphanumeric modes.

Service number 2, interrupt 10H (set cursor position), can also be used in conjunction with service number 9 to place a text string at any desired screen position, notwithstanding that there is no physical cursor in VGA graphics modes, and that the action of service number 2 is limited to selecting a screen location for the text display operation that follows. The invisible graphics mode cursor is sometimes called a *virtual cursor*.

#### Text Block Display

Programs that have frequent need to display text while in a graphics mode often need more display control than can be obtained by a virtual cursor and BIOS service number 9. One option is to develop a routine capable of displaying any number of text lines, starting at any screen position, and using any desired color available. A convenient way of storing the display parameters for the text message is in a header block preceding the message itself. The following procedure can be used for displaying a formatted text block:

```
;****************************************************************
;        procedure to display a formatted text block
;****************************************************************
;
TEXT_BLOCK PROC FAR
;
; Text message is format as follows:
; OFFSET:       CONTENTS:
;       0 ---- Screen row
;       1 ---- Screen column
;       2 ---- Color code
; DISPLAY CODES:
;       00H -- End of message
;       FFH -- End of screen line
; On entry:
;          DI -- Graphic display block
;
; Note: code assumes that write mode 0 is active
;
        MOV     DH,[DI]            ; Get screen row into DH
        INC     DI                 ; Bump pointer
        MOV     DL,[DI]            ; And column into DL
        MOV     BP,DI              ; Save address of screen column
                                   ; variable in BP
; Calculate complement to reverse rotation direction
;                               |  ST(0)    |   ST(1)   |   ST(2)
        FLD     CS:THREE_SIXTY  ;   360     |
        FILD    CS:ROT_ANGLE    ;    @      |    360    | -------|
        FSUB                    ; 360 - @   | --------  |
; Calculate sine and cosine of rotation angle and store
;                               |  ST(0)    |   ST(1)   |   ST(2)
        CALL    DEG_TO_RAD      ;  @ rads   | --------  |
        FLD     ST(0)           ;  @ rads   |  @ rads   | ------ |
        CALL    SINE            ;  sin @    |  @ rads   | ------ |
        FSTP    CS:SIN_@        ;  @ rads   | --------  |
        CALL    COSINE          ;  cos @    | --------  |
        FSTP    CS:COS_@        ; --------- |
; Routine to compute x' and y'
; x' = x cos @ - y sin @
;                               |  ST(0)    |   ST(1)   |   ST(2)
        FLD     CS:SIN_@        ;  sin @    | --------  |
        FLD     CS:CART_Y       ;   y       |  sin @    |  ----- |
        FMULP   ST(1),ST        ;  y sin @  | --------- |
        FLD     CS:COS_@        ;  cos @    |  y sin @  | ------ |
        FLD     CS:CART_X       ;   x       |  cos @    | y sin @|
        FMULP   ST(1),ST        ; x cos @   | y sin @   | ------ |
        FSUB    ST,ST(1)        ;   x'      | y sin @   |
        FSTP    ST(1)           ;   x'      | --------  |
; x' remains in the 8087 stack during computation of y'
; y' = y cos @ + x sin @
        FLD     CS:SIN_@        ;  sin @    |     x'    |
        FLD     CS:CART_X       ;    x      |   sin @   |    x'  |
        FMULP   ST(1),ST        ; x sin @   |     x'    | ------ |
        FLD     CS:COS_@        ;  cos @    |  x sin @  |   x'   |
        FLD     CS:CART_Y       ;   y       |   cos @   | x sin @|
        FMULP   ST(1),ST        ;  y cos @  |  x sin @  |   x'   |
        FADD                    ;    y'     |    x'     | ------ |
; Store x' and y' in original variables
        FSTP    CS:CART_Y       ;    x'     | --------  |
        FSTP    CS:CART_X       ; --------- |
        RET
ROTATE  ENDP
```

## 11.3 Operations on Text

Animated graphics applications often require some form of text display. If the text display functions in an application take place in separate screens from the graphics operations, then the programmer has the option of selecting a text mode and either using text output keywords in a high-level language or one of the text display functions available in the BIOS. However, if a graphics program must combine text and graphics on the same screen, the text display functions available to the programmer are more limited.

```
VIDEO_Y_UP      EQU       4      ; Upper screen limit
;
VIDEO_LIMITS    PROC    NEAR
; Test current display coordinates against video display limits
; defined in equates
; On entry:
;          BX = x coordinate of object
;          CX = y coordinate of object
; On exit:
;          carry set if object not displayable
;
; First test for x coordinate limits
        CMP     BX,VIDEO_X_RT   ; Test right limit
        JA      BAD_VID_RANGE   ; Go if x greater than right
                                ; limit
        CMP     BX,VIDEO_X_LF   ; Test left limit
        JB      BAD_VID_RANGE   ; Go if x smaller than left limit
; Now test for y coordinate limits
        CMP     CX,VIDEO_Y_DN   ; Test down limit
        JA      BAD_VID_RANGE   ; Go if y greater than down limit
        CMP     CX,VIDEO_Y_UP   ; Test up limit
        JB      BAD_VID_RANGE   ; Go if x smaller than left limit
; At this point object is within limits
        CLC                     ; Carry clear indicates limit OK
        RET
BAD_VID_RANGE:
        STC                     ; Carry set indicates limit error
        RET
VIDEO_LIMITS    ENDP
;****************************************************************
;
ROTATE  PROC    NEAR
; Rotate x and y coordinates of the first quadrant of a curve
;
; On entry:
;       CS:ROT_ANGLE = desired angle of rotation (in degrees)
;       CS:CART_X = x coordinate
;       CS:CART_Y = y coordinate
; On exit:
;       CS:CART_X = x' (rotated x coordinate)
;       CS:CART_Y = y' (rotated y coordinate)
;
; Formulas for obtaining rotated coordinates x' and y':
;       x' = x cos @ - y sin @
;       y' = y cos @ + x sin @
;
; Test for no rotation angle
        CMP     CS:ROT_ANGLE,0
        JNE     OK_ROTATE
        RET
OK_ROTATE:
```

### Subservice 5 — Check Button Press Status

Programs that do not use interrupts can check mouse button press status by calling subservice number 5 of the Microsoft mouse interface. The call is typically located in a polling loop. The calling program passes the button code in the BX register; the value of 0 corresponds to the left mouse button and the value of 1 to the right button. The call returns the button status in the AX register; bit 0 is mapped to the left mouse button and bit 1 to the right mouse button. A value of 1 indicates that the corresponding button is down. The BX register returns the number of button presses that have occurred since this call was last made or since a driver software reset (see subservice 0 earlier in this section). The CX and DX registers hold the *x* and *y* cursor coordinates of the screen position where the last press occurred. The following fragment shows a call to this subservice:

```
;****************************************************************
;                    button action handler
;****************************************************************
; The following routine calls service 5 of interrupt 33H to
; detect mouse press action on the mouse device
; If the right button was pressed execution is directed to the
; label RIGHT_BUT, if the left button was pressed execution is
; directed to the label LEFT_BUT
;**********************|
;  check left button   |
;**********************|
        MOV     AX,5            ; Service request to read
                                ; mouse button status
        MOV     BX,0            ; First test left button
        INT     33H             ; Mouse interrupt
; Number of button presses is returned in the BX register
        CMP     BX,0            ; Test for no presses
        JE      TEST_RIGHT_BUT  ; Not pressed. Test right button
; Code at this point should take the program action corresponding
; to one or more presses of the left mouse button
        .
        .
        .
; Execution should be allowed to fall through to the right button
; test routine
;**********************|
;  check right button  |
;**********************|
TEST_RIGHT_BUT:
        MOV     AX,5            ; Service request to read
                                ; mouse button status
        MOV     BX,1            ; Test right button
        INT     33H             ; Mouse interrupt
; Number of button presses is returned in the BX register
        CMP     BX,0            ; Test for no presses
        JE      END_BUTTON_RTN   ; Not pressed. End of routine
; Code at this point should take the program action corresponding
; to one or more presses of the right mouse button
        .
        .
        .
; Button press status processing ends at this label
END_BUTTON_RTN:
        .
        .
        .
```

## Subservice 11 — Read Motion Counters

The actual movement of the mouse-controlled icon is dependent on the state of two counters maintained by the mouse interface software. The Microsoft mouse interface at interrupt 33H stores the motion parameters in 1/200-in units called *mickeys*. The changes in the motion counters represent values from the last time the function was called. Subservice 11, of interrupt 33H, returns the values stored in the horizontal and vertical motion counters. The horizontal motion count is returned in the CX register and the vertical count in the DX register. The values are signed integers in 2's-complement form. A negative value in the horizontal motion counter indicates mouse movement to the left, while a negative value in the vertical motion counter indicates a movement in the upward direction. Both the vertical and the horizontal counters are automatically reset by the service routine.

The detection of mouse action can be by polling loops or by interrupts. Polling loops are often used in reading the motion counters so as to keep interrupt processing times to a minimum, specially considering that the Microsoft mouse interface does not allow the installation of more than one service routine. The processing inside a polling loop or a service routine takes place in similar fashion. The following fragment shows the structure of a basic mouse movement handler:

```
;****************************************************************
;                       mouse movement handler
;****************************************************************
; The following routine calls service 11 of interrupt 33H to
; detect horizontal or vertical movement of the mouse device
; If the movement is along the x axis (horizontal), execution is
; directed to the label H_MOVE. If the movement is along the y
; axis, execution is directed to the label Y_MOVE. If no change
; is detected in the motion counters, then execution is directed
; to the label NO_MOVE
;*********************|
;  service No. 11 of  |
;       INT 33H       |
;*********************|
        MOV     AX,11           ; Service request to read
                                ; motion counters
        INT     33H             ; Mouse interrupt
; CX = Horizontal mouse movement from last call to this service
; DX = vertical mouse movement from last call
        MOV     AL,CL           ; Horizontal counter to AL
        MOV     AH,DL           ; Vertical counter to AH
        CMP     AX,0            ; If AX is 0 then no mouse
        JNE     XORY_MOVE       ; Some movement detected
        JMP     NO_MOVE         ; Go if no movement
; At this point there is vertical or horizontal mouse movement
XORY_MOVE:
        CMP     CX,0            ; Test for no horizontal
        JE      Y_MOVE          ; Go to vertical movement test
;*********************|
;   horizontal move   |
;*********************|
; Code at this point moves the mouse icon according to the
; direction and magnitude of the value in the CX register
X_MOVE:
        PUSH    DX              ; Save vertical move counter
        .
        .
        .
        POP     DX              ; Restore vertical counter
; Once the horizontal movement is executed the code should fall
; through to the vertical movement routine. This takes care of
; the possibility of simultaneous movement along both axes
;*********************|
;    vertical move    |
;*********************|
; Code at this point moves the mouse icon according to the
; direction and magnitude of the value in the DX register
Y_MOVE:
        .
        .
        .
;*********************|
;   no movement       |
;*********************|
; This label is the routine's exit point
NO_MOVE:
        .
        .
        .
```

## Subservice 12 — Set Interrupt Routine

The user action on the mouse hardware can be monitored by polling or by interrupt generation, as is the case with most other input devices. Polling methods are based on querying the device status on a time-lapse basis; therefore polling routines as usually coded as part of execution loops. In the case of the mouse hardware the polling routine can check the motion counter registers and the button press and release status registers that are maintained by the mouse interface software. The services to read these registers are described later in this section.

The second and often preferred method of monitoring user interaction with the mouse device, particularly mouse button action, is by means of hardware interrupts. In this technique the program enables the mouse hardware actions that generate interrupts and installs the corresponding interrupt handlers. Thereafter, user action on the enabled hardware sources in the mouse automatically transfers control to the handler code. This frees the software from polling frequency constraints and simplifies program design and coding.

A typical application enables mouse interrupts for one or more sources of user interaction. For example, a program that uses the mouse to perform menu selection would enable an interrupt for movement of the trackball (or other motion detection mechanism) and another interrupt for the action of pressing the left mouse button. If the mouse is moved, the interrupt handler linked to trackball movement changes the screen position of the marker or icon according to the direction and magnitude of the movement. If the left mouse button is pressed, the corresponding interrupt handler executes the selected menu option.

Another frequently used programming method is to poll the mouse motion counters that store trackball movement and to detect button action by means of interrupts. This design reduces execution time inside the interrupt handler, which can be an important consideration in time-critical applications.

In the mouse interface software, the hardware conditions that can be programmed to generate an interrupt are related to an integer value called the *call mask*. Figure 13.3 shows the call mask bit map in the Microsoft mouse interface software. To enable a mouse interrupt condition the software sets the corresponding bit in the call mask. To disable a condition the call mask bit is cleared.

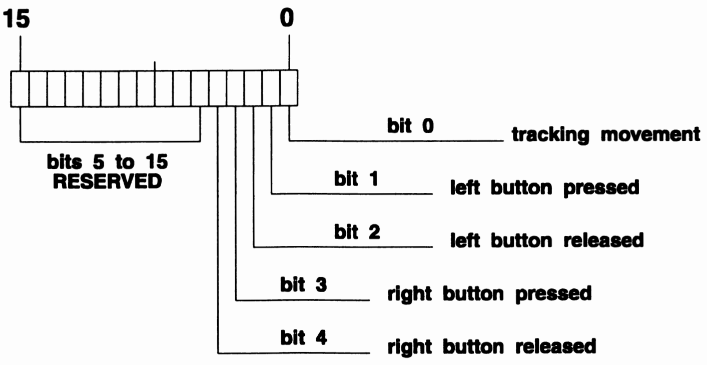


Figure 13.3 *Mouse Interrupt Call Mask Bitmap.*

Subservice number 12 of the mouse interface at interrupt 33H provides a means for installing an interrupt handler and for selecting the action or actions that generate the interrupt. The following fragment shows the necessary processing for enabling mouse interrupts on right and left button pressed.

```
; Select left mouse button pressed and right mouse button pressed
; as interrupt conditions and set address of service routine
; by means of mouse subservice number 12, interrupt 33H
;
; The code assumes that the interrupt handler is located in the
; program's code segment, at the offset of the label named
; MOUSE_ACTION
;
        CLI                     ; Interrupts off
        PUSH    ES              ; Save video buffer segment
        PUSH    CS              ; Program's segment
        POP     ES              ; to ES
        MOV     AX,12           ; Mouse service number 12
; Interrupt mask bit map:
;  15------------------- 5 4 3 2 1 0
;  |- these bits unused --| | | | | |___ Tracking movement
;                           | | | |_____ Left button pressed
;                           | | |_______ Left button released
;                           | |_________ Right button pressed
;                           |___________ Right button released
        MOV     CH,0            ; Unused bits
        MOV     CL,00001010B    ; Interrupt on left button and
                                ; right button pressed
        MOV     DX,OFFSET CS:MOUSE_ACTION ; Address of the
                                ; service routine
        INT     33H             ; Mouse interrupt
        POP     ES              ; Restore segment
        STI                     ; Interrupts on
        .
        .
        .
```

When the user's interrupt service routine receives control, the mouse interface software passes a condition code in the AL register that matches the call mask bit map (see Figure 13.3). In this manner the user's handler can determine which of the unmasked conditions actually generated the interrupt. An interrupt condition bit is set when the corresponding condition originated the interrupt. For example, if the conditions that originate the interrupt are the left or right mouse buttons pressed, then the program can test the state of bit number 1 (see Figure 13.3) to determine if the interrupt was caused by the left mouse button. If not, the code can assume that it was caused by the user pressing the right mouse button, since only these two conditions are active.

A characteristic of service number 12 or the Microsoft mouse interface is that only one interrupt handler can be installed. If two consecutive calls are made to this service, even if the call mask settings enable different bits, the address in the latest call replaces the previous one. Therefore it is not possible to install more than one service routine by means of this service. On the other hand, service number 24 allows the installation of more than one service routine, each one linked to a different interrupt cause. However, this service operates only when the Shift, Ctrl, or Alt keys are held down while the mouse action is performed. In addition, in several non-Microsoft versions of the mouse interface software this service does not perform as documented. For these reasons it is not considered in this book.

## 13.4 Cursor in VGA Graphics Mode

Not all data input into a graphics application can be entered by means of an analog device such as a mouse, a joystick, or a puck. Very often a program executing in a VGA graphics mode requires that the user enter digital data, such as names, numbers, or codes. However, the cursor, which is the conventional positioning mechanism for data input, is available only in the VGA text modes. This is due to the fact that the VGA cursor hardware is inactive in the standard or nonstandard *All-Points Addressable* (APA) modes.

Implementing a cursor that functions in VGA graphics mode is a matter of contriving a way of displaying a flashing marker that signals the current input point, of displaying the input character, and of updating the marker position according to the user's interaction. The IBM extended character set includes several graphics objects that can be used as a cursor character, but a simple underscore symbol from the ASCII character set usually suffices for this purpose.

The code must also find a way of making the cursor symbol flash on the screen. This requires a timed beat for successively displaying and erasing the cursor. The timer beat can be obtained by intercepting the system timer interrupt, as discussed in Chapter 12. Finally, the code must keep track of the input position, which implies recognizing some editing keys such as the backspace.

### 13.4.1 Intercepting the System Timer Interrupt

The system timer interrupt was discussed in Chapter 12. The following code fragment shows a simple intercept of interrupt 1CH in order to obtain a timed beat at the rate of 18.2 times per second:

```
;****************************************************************
;        install timer beat intercept for graphics cursor
;****************************************************************
        CLI
;*********************|
;  get address of old |
;       handler       |
;*********************|
; Uses MS-DOS service 53 of interrupt 21H to obtain the original
; address for INT 1CH
        PUSH    ES              ; Save segment
        MOV     AH,53           ; Service request code
        MOV     AL,1CH          ; Code of vector desired
        INT     21H
; ES - Segment address of installed interrupt handler
; BX - Offset address of installed interrupt handler
        MOV     CS:OLD1C_ADD,BX ; Store offset in variable
        MOV     CS:OLD1C_SEG,ES ; Store segment base
        POP     ES              ; Restore segment
;*********************|
; install new handler |
;*********************|
; Take over timer tick at INT 1CH Using DOS service 37 of
; interrupt 21H
        MOV     AH,37           ; Service request number
        MOV     AL,1CH          ; Interrupt to be intercepted
        LEA     DX,CS:HEX1C_INT ; Pointer to handler
        PUSH    DS              ; Save data segment
        PUSH    CS
        POP     DS
        INT     21H             ; MS-DOS interrupt
        POP     DS              ; Restore DS
        STI
;****************************************************************
;                end of interrupt installation
;****************************************************************
        .
        .
        .
```

### 13.4.2 The Timer Interrupt Handler

Once the timer interrupt intercept is implemented (see Section 13.4.1) the program receives control every 18.2 times per second at the corresponding label. Therefore, the programmer must code a handler routine at the intercept label. In implementing a graphics mode cursor the beat of 18.2 times per second provides too fast a flashing rate for the cursor object. The handler can slow down the pulse by skipping some of the intercepts. In the handler routine listed below the code keeps track of intercepts in the variable named TOGGLE_CTRL. This variable takes values from 0 to 9. When the value 3 is reached, the cursor is turned on. When the value 6 is reached, the cursor is turned off. The result is a pleasant flashing rate of the cursor.

```
;****************************************************************
;               interrupt 1CH intercept routine
;****************************************************************
```

```
; Code segment data
TOGGLE_CTRL     DB      0       ; Control for cursor ON/OFF
CUR_ONOFF       DB      0       ; Cursor ON or OFF switch
                                ; 1 = display cursor
                                ; 0 = skip intercept
CUR_ATTR        DB      0       ; Current attribute
OLD1C_ADD       DW      0       ; Storage for old interrupt 1CH
OLD1C_SEG       DW      0
;
HEX1C_INT       PROC    FAR
; Interrupts on
        STI                 ; Reenable interrupts
;**********************|
;  test ON/OFF switch  |
;**********************|
        CMP     CS:CUR_ONOFF,1  ; Test for ON mode
        JE      CURSOR_ACTIVE   ; Go if 1
        IRET
;**********************|
;     save context     |
;**********************|
; Save all registers used by the interrupt intercept routine
; including the ES segment register
CURSOR_ACTIVE:
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSH    DI
;**********************|
;  test toggle byte    |
;**********************|
; The byte at TOGGLE_CTRL serves to determine the intercept used
; to turn the cursor ON and OFF
        CMP     CS:TOGGLE_CTRL,3  ; Is value = 3?
        JE      DISPLAY_CUR       ; Go if live iteration
        CMP     CS:TOGGLE_CTRL,6  ; Is value = 6?
        JE      ERASE_CUR         ; Go if live iteration
; Manage cursor control byte
        INC     CS:TOGGLE_CTRL;
        CMP     CS:TOGGLE_CTRL,9
        JB      CTRL_IN_RANGE     ; Control is in range
; Reset control byte
        MOV     CS:TOGGLE_CTRL,0  ; Restart counter
CTRL_IN_RANGE:
        JMP     CUR_EXIT
DISPLAY_CUR:
        MOV     AL,'_'            ; Graphic cursor character
CURSOR_RTN:
        INC     CS:TOGGLE_CTRL    ; Bump control byte
        MOV     AH,9              ; Service request number
        MOV     BL,CS:CUR_ATTR    ; Cursor attribute
        MOV     BH,0              ; Display page zero
        MOV     CX,1              ; One character
        INT     10H               ; BIOS service
        JMP     CUR_EXIT          ; Go
;*********************|
;   erase cursor      |
;*********************|
ERASE_CUR:
        MOV     AL,20H            ; Space
        JMP     CURSOR_RTN
;*********************|
;     exit routine    |
;*********************|
CUR_EXIT:
        POP     DI                 ; and general registers
        POP     SI
        POP     DX
        POP     CX
        POP     BX
        POP     AX
; Exit from new service routine to old service routine
        IRET
HEX1C_INT       ENDP
```

Notice that the listed handler displays and erases the cursor character by means of BIOS service number 9, of INT 10H. The handler takes advantage of the fact that service number 9 uses the current position of the system cursor to display the character. This simplifies programming by allowing the use of the BIOS cursor control services.

### 13.4.3 Keyboard Input Routine

In order to use the mechanism of a hardware cursor in a graphics mode it is necessary to code a routine that turns on the cursor, obtains the user input echoing the typed characters, performs the necessary input editing functions, and turns off the cursor when input concludes. A simple set of input control keys is based on using the <Enter> key to terminate the input phrase, <Esc> to abort input, and <Backspace> to erase the preceding character and move the cursor back one screen character.

Several auxiliary procedures facilitate the operation of the input routine. One procedure sets the system cursor location, which is used by the intercept routine listed in Section 13.4.2. Another procedure displays the user's input character, and two other auxiliary procedures turn the graphics mode cursor on and off.

```
;****************************************************************
;              graphics cursor input procedure
;****************************************************************
```

```
APA_INPUT       PROC    NEAR
; General keyboard input routine for VGA graphics mode with
; graphics cursor display
;
; Control codes and exit codes:
;               0DH <Enter> .... Exit with carry clear
;               011BH <Esc> .... Exit with carry set
;               08H <Backspace>. Erase character and backspace
;                                cursor
; On entry:
;          DI -> storage buffer
;          SI -> Input format area:
;                 Offset 0 = screen start row
;                 Offset 1 = screen start column
;                 Offset 2 = total characters allowed
;                 Offset 3 = display attribute
; On exit:
;          Text is stored in buffer by DI
;                 DI/SI -> start of character buffer
;                 CX = total characters input
;                 carry clear
;
        PUSH    DX                  ; Save entry DX
        MOV     DH,[SI]             ; Get display row
        INC     SI                  ; Bump to offset 2
        MOV     DL,[SI]             ; Get display column
        CALL    SET_CUR             ; Local procedure
        INC     SI                  ; Bump to offset 3
        MOV     CL,[SI]             ; Set up counter
        MOV     CH,CL               ; Copy start value in CH
; Get current display attribute
        INC     SI                  ; Pointer to attribute
        MOV     AL,[SI]             ; Read attribute in block
        MOV     CS:CUR_ATTR,AL      ; Store in CS variable
; Save entry value of buffer pointer in SI
        MOV     SI,DI
; Turn on graphics cursor
        CALL    APA_CUR_ON          ; Local procedure
GET_KEY:
        CALL    KBR_WAIT            ; Get character
;******************|
; input processing |
;******************|
        CMP     AL,0DH              ; <Enter> key
        JNE     NOT_0DH
        JMP     KBR_EXIT_0          ; Take exit
NOT_0DH:
        CMP     AX,011BH            ; <Esc> key
        JNE     NOT_ESC
        JMP     KBR_EXIT_1          ; Take exit
NOT_ESC:
        CMP     AL,08H          ; <Backspace> key
        JNE     NOT_BAK
;*****************|
;    backspace    |
;*****************|
; Test for cursor at start of buffer
        CMP     CH,CL           ; Test present count with start
                                ; value
        JE      GET_KEY         ; Ignore backspace at start
                                ; position
; Turn off and erase cursor
        CALL    APA_CUR_OFF     ; Local procedure
; Execute backspace
        DEC     DI              ; Buffer pointer
        DEC     DL              ; Display column counter
        CALL    SET_CUR
        INC     CL              ; Adjust maximum characters count
        MOV     AL,' '          ; Blank space
        CALL    SHOW_APA        ; Display a blank
        MOV     [DI],AL         ; and put blank in buffer
; Restore graphics cursor
        CALL    APA_CUR_ON      ; Local procedure
        JMP     GET_KEY         ; Continue
;******************|
; test for errors  |
;******************|
NOT_BAK:
; Test for invalid input (less than 20H or more than 79H)
        CMP     AL,20H
        JL      GET_KEY         ; Illegal, too small
        CMP     AL,79H
        JG      GET_KEY         ; Illegal, too large
; Test for buffer full
        CMP     CL,0            ; CL is counter
        JNZ     DISPLAY_IT
        JMP     GET_KEY         ; Buffer is full
;******************|
;  display char.   |
;******************|
DISPLAY_IT:
        CALL    SHOW_APA
; Store it in buffer
        MOV     [DI],AL
;******************|
;  bump pointers   |
;******************|
        INC     DI              ; Bump buffer pointer
        INC     DL              ; Bump display column
        CALL    SET_CUR
        DEC     CL              ; Character counter
        JMP     GET_KEY         ; Continue
```

```
;
KBR_EXIT_0:
        PUSH    SI              ; Save start of buffer
        SUB     DI,SI           ; Get total input
        MOV     CX,DI           ; into counter
        POP     DI              ; Restore buffer start pointer
        CLC                     ; <Enter> key exit, no carry
        POP     DX              ; Restore entry DX
        CALL    APA_CUR_OFF     ; Turn off graphics cursor
        RET
KBR_EXIT_1:
        PUSH    SI              ; Save start of buffer
        SUB     DI,SI           ; Get total input
        MOV     CX,DI           ; into counter
        POP     DI              ; Restore buffer start pointer
        STC                     ; <Esc> key exit, carry set
        POP     DX              ; Restore entry DX
        CLC                     ; No error detection
        CALL    APA_CUR_OFF     ; Turn off graphics cursor
        RET
APA_INPUT       ENDP
;
;****************************************************************
;         support procedures for graphics cursor input
;****************************************************************
SET_CUR         PROC    NEAR
; Set system cursor to a row and column coordinate
; On entry:
;         DH = screen row (range 0 to 24)
;         DL = screen column (range 0 to 79)
; On exit:
;         carry clear if no error
;
        PUSH    AX              ; Save entry registers
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,2            ; Service request number
        MOV     BH,0            ; Assume display page No. 0
        INT     10H
        POP     DX              ; Restore registers
        POP     CX
        POP     BX
        POP     AX
        RET
SET_CUR ENDP
;***************************************************************
;
SHOW_APA        PROC    NEAR
; Display the character in AL at the current cursor position
; On entry:
```

```
;          AL = character
; On exit:
;          carry clear
;
        PUSH    AX              ; Save caller's registers
        PUSH    BX
        PUSH    CX
        MOV     AH,9            ; Service request number
        MOV     BL,CS:CUR_ATTR  ; Current attribute
        MOV     BH,0            ; Display page zero
        MOV     CX,1            ; One character
        INT     10H
        POP     CX              ; Restore caller's registers
        POP     BX
        POP     AX
        RET
SHOW_APA        ENDP
;****************************************************************
;
APA_CUR_ON      PROC    NEAR
; Procedure to activate the graphics cursor
        MOV     CS:CUR_ONOFF,1          ; Switch ON
        RET
APA_CUR_ON      ENDP
;****************************************************************
;
APA_CUR_OFF     PROC    NEAR
; Procedure to deactivate the graphics cursor
        PUSH    AX              ; Save context
        PUSH    BX
        PUSH    CX
        MOV     CS:CUR_ONOFF,0          ; Switch ON
        MOV     AL,' '          ; Erase cursor
        MOV     AH,9            ; Service request number
        MOV     BL,00001100B    ; Red
        MOV     BH,0            ; Display page zero
        MOV     CX,1            ; One character
        INT     10H             ; BIOS service
        POP     CX              ; Restore context
        POP     BX
        POP     AX
        RET
APA_CUR_OFF     ENDP
```

# Bibliography

## Books and Technical Manuals

Adams, Lee. *High-Speed Simulation and Animation for Microcomputers.* Blue Ridge Summit, PA: TAB Books, 1987.

Arnheim, Rudolf. *Art and Visual Perception.* Berkeley, CA: University of California Press, 1974.

Artwick, Bruce A. *Applied Concepts in Microcomputer Graphics.* Englewood Cliffs, NJ: Prentice-Hall, 1984.

—— *Microcomputer Displays, Graphics, and Animation.* Engelwood Cliffs, NJ: Prentice-Hall, 1985.

Burks, A.W. (Ed.). *Essays on Cellular Automata.* Urbana: University of Illinois Press, 1970.

Codd, E. F. *Cellular Automata.* ACM Monograph Series. New York: Academic Press, 1968.

Conrac Corporation. *Raster Graphics Handbook.* New York: Van Nostrand Reinhold, 1985.

Doty, David B. *Programmer's Guide to the Hercules Graphics Cards.* Reading, MA: Addison-Wesley, 1988.

Enderle, G., K. Kansy, and G. Pfaff. *Computer Graphics Programming.* Berlin: Springer, 1984.

Ferraro, Richard F. *Programmer's Guide to EGA and VGA Cards.* Reading, MA: Addison-Wesley, 1988.

Fox, David, and Michael Waite. *Computer Animation Primer.* New York: McGraw-Hill, 1984.

Harrington, Steven. *Computer Graphics: A Programming Approach.* New York: McGraw-Hill, 1983.

Harris, Dennis. *Computer Graphics and Applications.* London: Chapman and Hall Computing, 1984.

Hearn, Donald, and M. Pauline Baker. *Computer Graphics.* Englewood Cliffs, NJ: Prentice-Hall, 1986.

Hoggar, S. G. *Mathematics for Computer Graphics.* Cambridge, England: Cambridge University Press, 1992.

IBM Corporation. *Technical Reference, Personal Computer.* Boca Raton, FL: IBM, 1984.

—— *Personal System/2 Hardware Interface Technical Reference - Video Subsystems.* Boca Raton, FL: IBM, 1992.

—— *Personal System/2 and Personal Computer BIOS Interface Technical Reference.* Boca Raton, FL: IBM, 1987.

—— *Technical Reference, Options and Adapters.* Boca Raton, FL: IBM, 1986.

——— *Technical Reference, Personal System/2.* Boca Raton, FL: IBM, 1987.

——— *Technical Reference, Options and Adapters.* XGA Video Subsystem. Boca Raton, FL: IBM, 1986.

——— *XGA Video Subsystem Hardware User's Guide.* Boca Raton, FL: IBM, 1990.

Intel Corporation. *80286 and 80287 Programmer's Reference Manual.* Santa Clara, CA: Intel, 1987.

——— *80386 Programmer's Reference Manual.* Santa Clara, CA: Intel, 1986.

——— *iAPX 86/88, 186/188 User's Manual (Programmer's Reference).* Santa Clara, CA: Intel, 1987.

Kepes, Gyorgy, (Ed.). *Sign, Image, Symbol.* New York: George Braziller, 1966.

Kliewer, Bradley Dyck. *EGA/VGA A Programmer's Reference Guide.* New York: McGraw-Hill, 1988.

Magnenat-Thalmann, Nadia, and Daniel Thalmann. *Computer Animation: Theory and Practice.* Tokyo: Springer-Verlag, 1985.

Mandelbrot, Benoit B. *The Fractal Geometry of Nature.* N.Y: W.T. Freeman and Co., 1982.

Mealing, Stuart. *The Art and Science of Computer Animation.* Oxford, England: Bath Press, 1992.

Microsoft. *Microsoft Mouse Programmer's Reference.* Redmond, WA: Microsoft Press, 1989.

Myers, Roy E. *Microcomputer Graphics.* Reading, MA: Addison-Wesley, 1982.

Pimentel, Ken, and Kevin Teixeira. *Virtual Reality: Through the New Looking Glass.* New York: McGraw-Hill, 1993.

Pokorny, Cornel K., and Curtis F. Gerald. *Computer Graphics: The Principles Behind the Art and Science.* Irvine, CA: Franklin, Beedle & Associates, 1989.

Ralston, Anthony, and Chester L. Meek. *Encyclopedia of Computer Science.* New York: Mason and Charter, 1983.

Richter, Jake. *Power Programming ... the IBM XGA.* New York: MIS Press, 1992

Richter, Jake, and Bud Smith. *Graphics Programming for the 8514/A.* Redwood City, CA: M & T Books, 1990.

Rietman, Edward. *Creating Artificial Life: Self Organization.* Blue Ridge Summit, PA: TAB Books, 1993.

Rimmer, Steve. *Bit-Mapped Graphics.* New York: McGraw-Hill, 1990.

——— *The Graphics File Toolkit.* Reading, MA: Addison-Wesley, 1992.

——— *Supercharged Bitmapped Graphics.* New York: McGraw-Hill, 1992

——— *SuperVGA Graphics Programming Secrets.* New York: McGraw-Hill, 1993

Rogers, David F. *Procedural Elements for Computer Graphics.* New York: McGraw-Hill, 1985.

Salmon, Rod, and Mel Slater. *Computer Graphics, Systems & Concepts.* London: Addison-Wesley, 1987.

Sanchez, Julio. *Graphics Design and Animation on the IBM Microcomputers.* Engelwood Cliffs, NJ: Prentice-Hall, 1990.

Sanchez, Julio, and Maria P. Canton. *IBM Microcomputers: A Programmer's Handbook.* New York: McGraw-Hill, 1990.

——— *Graphics Programming Solutions.* New York: McGraw-Hill, 1993

——— *High Resolution Video Graphics.* New York: McGraw-Hill, 1994

——— *Programming Solutions Handbook for IBM Microcomputers.* New York: McGraw-Hill, 1991.

Sproull, Robert F., W. R. Sutherland, and Michael K. Ullner. *Device Independent Graphics.* New York: McGraw-Hill, 1985.

Stevens, Roger T. *Fractal Programming in C.* Redwood City, CA: MT Books, 1989.

Sutty, George, and Steve Blair. *Advanced Programmer's Guide to SuperVGAs.* New York: Simon & Schuster, 1990.

VESA. *Super VGA BIOS Extension, June 2, 1990.* San Jose, CA: VESA, 1990.

——— *Super VGA Standard, Version 1.2, October 22, 1991.* San Jose, CA: VESA, 1991.

——— *XGA Extensions Standard, Version 1.0, May 8, 1992.* San Jose, CA: VESA, 1992.

Video Seven. *Video Seven VGA Programmer's Reference Manual.* Fremont, CA: Headland Technology Inc., 1991.

Watt, Allan, and Mark Watt. *Advanced Animation and Rendering Techniques, Theory and Practice.* Wokingham, England: ACM Press, 1992.

Wilton, Richard. *Programmer's Guide to PC & PS/2 Video Systems.* Redmond, WA: Microsoft Press, 1987.

## Periodicals and Other References

Abrash, Michael. "256-Color VGA Animation." *Dr. Dobb's Journal.* No. 180, August, 1991.

——— "Mode X: 256-Color VGA Magic." *Dr. Dobb's Journal.* No. 178, July, 1991.

——— "More Dirty (Dirtier?) Rectangles." *Dr. Dobb's Journal.* No. 197, February, 1993.

——— "More Undocumented 256-Color VGA Magic." *Dr. Dobb's Journal.* No. 179, August, 1991.

——— "Yet Another Animation Method.." *Dr. Dobb's Journal.* No. 196, January, 1993.

Demetrescu, Stefan. "Moving Pictures." *BYTE*, vol. 10, No. 12, November, 1985.

Gomez, J. E. *Comments on Event Driven Animation.* SIGGRAPH Course Notes 10, 1987.

# Index

## ABOUT THE AUTHORS

JULIO SANCHEZ is an associate professor of computer science at Montana State University, Northern.

MARIA P. CANTON is the president of Skipanon Software Company, a software development and consulting firm in Great Falls, Montana. Together they have authored several books on computer programming, mathematics, and graphics for McGraw-Hill, including *Graphics Programming Solutions*, *High Resolution Video Graphics*, *Numerical Programming on the 387, 486, and Pentium*, and *PC Programmers Handbook*, Second Edition.

$39.95 U.S.A.

Graphics

**The *definitive*, practical reference for PC animation programmers**

# Computer Animation
## Programming Methods & Techniques

Written by programmers *for* programmers, *Computer Animation* outlines the methods and problem-solving techniques you need to produce state-of-the-art animation on the PC. Packed with code samples for the fundamental device drivers and primitives of VGA, SuperVGA, and XGA Video systems, this comprehensive, detailed guide includes the very latest advances in video hardware technology. Key topics include:

Graphical Image Structures • Bitmap Acquisition and Encoding • Animation in VGA Graphics • Device Drivers for VGA Standard Modes • Drivers and Primitives for VGA Mode X • XGA Graphics and Animation • SuperVGA Graphics and Animation • Time-Pulse Animation • User-Animated Objects • Multiple Page Techniques • Microfeedback from Interactive Objects • and Much More

Whether you're developing applications for computer games, high-tech simulators, or event modeling in science and engineering, this outstanding reference offers the hands-on programming guidance needed to get the job done right.

ISBN 0-07-054964-8

90000

9 780070 549647

Cover Design: Trent Truman, Hagerstown, MD
Cover Background: Imtek Imagineering/Masterfile

**Computing McGraw-Hill**
1221 Avenue of the Americas
New York, NY 10020