BANTAM COMPUTER BOOKS

THE PROGRAMMING PERFORMANCE LIBRARY

ATARI ST APPLICATION PROGRAMMING

LAWRENCE J. POLLACK AND ERIC J.T. WEBER

DATATECH PUBLICATIONS

Atari ST
Application Programming

Atari ST Application Programming

Lawrence J. Pollack
Eric J.T. Weber

Datatech Publications

BANTAM BOOKS
TORONTO • NEW YORK • LONDON • SYDNEY • AUCKLAND

Atari ST Application Programming
A Bantam Book / Sept 1987

ISBN 0-553-34397-1

Published simultaneously in the United States and Canada

Bantam Books are published by Bantam Books, Inc. Its trademark, consisting of the words "Bantam Books" and the portrayal of a rooster, is Registered in U.S. Patent and Trademark Office and in other countries. Marca Registrada. Bantam Books, Inc., 666 Fifth Avenue, New York, New York 10103.

PRINTED IN THE UNITED STATES OF AMERICA

FG 0 9 8 7 6 5 4 3 2 1

Table of Contents

CHAPTER ONE

A Map of TOS

The disk drive(s), screen, keyboard, circuitry, and other components of a computer do nothing without instructions. The software instructions that permit the different hardware parts of a computer to interact are known as operating systems. Operating systems control the hardware functions that a computer must perform: system timing, program initiation, command interpretation, video display, keyboard interpretation, communication with external devices (such as printers, disk drives, and modems), and memory management.

Designing an operating system that works is not a trivial task because of the complexity of the functions involved. To facilitate this task, an operating system is usually broken into modules. Most operating systems are configured in modules that generally include a Disk Operating System (DOS), a Basic Input/Output System (BIOS), and some extended functions for providing easy access to the special hardware features built into the computer. DOS usually handles routines for manipulating data on disks, i.e., reading, writing, deleting, and organizing files. BIOS provides routines for primary communication with the keyboard, screen, and other external devices. The extended functions can include graphics, sound, and memory management.

The operating system for the Atari ST computer is called TOS. Like most other operating systems, TOS is modularly designed and consists of several components. This chapter explains what is included in TOS, the function of each component of the operating system, and the relationships between the components. TOS includes all of the basic parts, or modules, of an operating system listed above. However, it organizes them into three basic components: GEM, XBIOS, and the

Line A handler. Each of these components governs different functions of the system, but in some cases there is an overlap in function. The component with the most obvious difference is the Graphics Environment Manager (GEM); therefore, it is the first module examined.

GEM

Introduction

A graphics environment is exactly what it sounds like: one that uses pictures rather than text to represent specific tasks the computer can perform. Examples include a filing cabinet representing the floppy disk drives that store files and a trash can representing the place to put data no longer needed. In general, an environment manager keeps track of what is shown on the screen. A *graphics* environment manager, therefore, keeps in order the special features the graphics environment uses: icons, windows, and other functions.

Using a picture language for communication dates back to the ancient Egyptians, but its use in the electronic communications world is a far more recent development. In fact, most computers still use a text-based system where the user must type commands on the keyboard rather than pointing with a mouse. Obviously, a graphics-based computer program is much easier for a beginning computer user to interact with.

If it is such a user-friendly way of doing business with a computer, why don't all computers utilize a graphics environment? It's a matter of convenience and economics. The hardware and software required to run a text environment cost much less and are much simpler to develop than that needed for a graphics environment. The cost savings occur primarily in the software end of things because of the time involved in development. Writing a program for a text environment merely requires sending text to a display and reading text from a keyboard, which is a relatively simple interface. A graphics environment, on the other hand, must be able to accept text and manipulate many complex images. It has to handle menus, windows, mouse movement and input—clicks and drags, icons, and other images. Each of these also contains several complex structures, such as a size box and title bar for a window, which must also be addressed by the environment manager.

The Xerox Palo Alto Research Center (PARC) first developed a graphics environment using a mouse, menus, windows, and icons. Unfortunately, at that time, the hardware allowing graphics displays was still quite expensive; this prevented the graphics environment from being pursued commercially.

Apple Computer designed the Lisa and then the Macintosh to use a graphics-based interface—one with the ability to store the various components of the graphics environment in libraries for easy access by outside programs. This heralded a new beginning for user-computer communications.

The best design aspect of the Macintosh was that its environment routines were built into the computer. This allowed programmers to access the routines from their programs rather than having to write thousands of lines of code to handle windows, menus, the mouse, and so on.

GEM and the Macintosh environments are similar in that they both provide graphics environments in which to program. However, GEM goes beyond the Macintosh environment in an important aspect: GEM is a portable environment.

Digital Research designed GEM for implementation on many different types of computers, for example, the Atari ST, IBM PC, and IBM PC compatibles. This means that a program written for one computer using GEM can be moved to a different computer without changing the logic of the program. If the program were initially written in a portable language such as C, virtually no changes would be needed. You could simply transfer the source code to the desired computer system and then recompile the program.

GEM not only affords portability among different computers but also provides hardware independence for output. Basically, a computer communicates to the user by providing feedback through some output device, usually a video screen or printer. Each output device has different characteristics for its display, including line width, dot size, and graphics capabilities. To translate a display from the screen onto paper, the data in the display must be reformatted to accommodate the differing characteristics of the paper device. Without GEM a programmer would need to write an output routine for each device to account for these differences. However, GEM supplies generic routines that take care of this translation. These routines utilize pieces of software called device drivers.

Each device driver contains a description of the device's specific characteristics. When a programmer runs a program written using GEM, the GEM routines access the appropriate device drivers. These convert the generic GEM routine information into device-specific output. This means that you can write a single function to draw a series of circles using GEM routines. That function can then be used to display the circles on a screen, a graphics printer, or a plotter without changes to the program. The usefulness of device independence should be fairly obvious. This function of the ST is examined further in this chapter when the GEM VDI is discussed.

The GEM interface has a two-fold function. First, GEM is an inter-

face between the user and the computer. It can make computer operations much easier, particularly for a novice, by providing windows, menus, and simple icons in place of text. Second, GEM is an interface between the programmer and the available hardware. With GEM, the time it takes to write user interfaces for the variety of hardware currently available greatly decreases, and productivity increases with the availability of a standardized interface such as GEM.

The Facets of GEM

GEM, like most operating systems, is written in modules. In fact, GEM could actually be used in place of TOS as the basic operating system for a computer. It supports a Disk Operating System (GEMDOS), a Virtual Device Interface (VDI), and an Application Environment Services (AES), which collectively contain all the information necessary to drive the computer. TOS contains functions and routines specific to the Atari ST's special features. This is why TOS is used with GEM.

GEMDOS

The Graphics Environment Manager Disk Operating System (GEMDOS) is the first module. GEMDOS consists of a Basic Input/Output System (BIOS), a disk file handler, and some useful system functions. The BIOS provides routines for disk access and primary communication with the keyboard, screen, and other external devices. The disk file handler keeps data organized on disks. GEM supports a hierarchical file structure using a primary (root) directory and folders (subdirectories) much like UNIX and MS-DOS. The system functions provided under GEMDOS include disk access, date and time setting, memory management, and process initiation and termination.

The VDI

The second module of GEM is the Virtual Device Interface, referred to as the VDI. The VDI is a standard graphics environment in which to write a graphics application. This means that the same VDI functions can be used with a video display, graphics printer, plotter, or any other graphic device. This happens through a combination of operations the VDI performs. The VDI consists of two parts: the Graphics Device Operating System (GDOS) and the device drivers and face files. These two parts work together to make possible the device-independent operations of GEM.

The GDOS provides the device-independent interface that allows you to use a standard set of functions for generating graphic images without being concerned about what device the image will be drawn on. The GDOS describes your programmed image to a coordinate system that corresponds to the "ideal" graphic device. This ideal device is simply a standard set of plotting points that can be used to correspond to the actual plotting points of a specific device. When you wish to output the image to an actual device, the image description is mapped onto the device's coordinate system through the device drivers. The device driver takes care of producing the proper image using the device's capabilities. If a particular feature is not provided by a device, the device driver will try to emulate that feature as closely as possible. For instance, some printers cannot provide italicized letters and underline the desired text instead. The face files contain a description of a particular alphabet font that can be easily translated to any device. The VDI maintains GEM's device-independent capability by allowing the same VDI functions to be used with any graphic device and still produce approximately the same image.

The AES

The Application Environment Services (AES) is a collection of *libraries* that allow a programmer to utilize the various graphic images to provide a standard user interface. A library is a set of commands that have been grouped together. The AES manages graphics input in much the same way as the VDI manages graphics output. In other words, the libraries of the AES contain functions for icons, drop-down menus, dialog boxes, alert messages, windows, and mouse control from a user. The GEM desktop, the workspace provided by the computer after boot up, is an example of a GEM application that uses the AES.

In addition to the subroutine libraries, the AES contains a limited multitasking kernel and dispatcher, a shell, a desk accessory buffer, and a menu/alert buffer. The subroutine libraries contain the functions for windowing, controlling the mouse, displaying system and error messages, and drawing AES objects on the screen. These functions are the elements of the AES most useful to a programmer; therefore, they are the focus of discussion of the AES.

Briefly then, the three portions of GEM are GEMDOS, VDI, and AES. These three components interact to manage the graphics-based environment the Atari uses and to make it device-independent. Before you learn to use GEM in applications programming, two additional parts of the operating system need to be explained.

The Line A Handler

The second component of TOS to be examined is the Line A Handler. This is a set of sixteen functions that provide primitive routines for graphics output to the video display. These functions include screen initialization, put and get pixels, line drawing, rectangle fill, polygon fill, bit map block transfers, show and hide mouse, sprite manipulation, and other miscellaneous functions. The Line A routines provide extremely fast execution of basic graphic primitives. In fact, the VDI uses the Line A functions to draw on the screen. Unfortunately, the Line A routines are only accessible from the assembly language level. Because this book deals primarily with programming the ST in C, only a brief coverage of these routines is given in Chapter 2.

The XBIOS

The basic input/output functions of the Atari ST could be handled by GEM through GEMDOS. However, the operating system provided by GEM is designed to provide hardware independence. This means that the operating system has to be a generic one that does not include special features of a given machine, such as sound and MIDI (musical instrument device interface). Also, because GEM uses device drivers to translate its generic routines into device-specific information, it is slower and less efficient than writing a routine to directly manipulate a device. To overcome this slight drawback, Atari has written the eXtended Basic Input/Output System (XBIOS). Written specifically for use with Atari hardware, XBIOS provides access to the special features of the machine. Of course, GEM and XBIOS occasionally overlap in functionality. For example, you can set the date and time with either GEM or XBIOS. In general, XBIOS provides access to machine-specific functions like sound, keyboard translation tables, MIDI, and color settings.

What To Use

Atari has supplied a very basic operating system, TOS, for a programmer to use in accessing the ST's hardware. GEM, on the other hand, is a generic operating system with graphic support that allows access to many different hardware systems. Either of these systems could be used to write a program for the ST. TOS and XBIOS programs are

very text-oriented and their operations (for example, character input/output and reading/writing to disk by sectors) work at a relatively low level. Working at a much higher level, GEM software provides a more natural view of how a program needs to be organized. The drawback to GEM is that it does not provide access to all of the ST's features such as sound and disk formatting. The question you are probably asking is "Which routines do I use?"

The answer is "It depends." GEM is a well-documented, consistent system that provides an excellent user interface. Throughout this book, GEM is used whenever possible. However, for some functions you need to use the less flexible TOS to access special ST hardware. For instances where you need to use Atari XBIOS functions, a note is made that these routines are not included in GEM.

Using GEM

Programs that use a graphics-based environment are far easier for a first-time user to interact with. Thanks to the advances of PARC, Apple, and Atari, they are also simpler for a programmer to produce. The ability to include library graphics routines that provide easy user interaction can save you literally hundreds of hours of work. Additionally, GEM's device-independent capabilities allow you to port programs intact from system to system and reproduce images on a variety of output devices without providing individual device instructions.

In order to be device-independent, GEM's VDI is divided into two distinct sections, the GEM interface and the device drivers. The interface takes a program's output and converts it to a generic internal representation. The device drivers can then take this internal representation and convert it to instructions for a specific output device. The same internal representation is used for all devices; saving this representation in a file frees you from having to recreate the image each time it is to be output. Saving is done through the use of metafiles discussed in Appendix A.

The GEM interface can be accessed in two ways: through assembly language or through C function calls. Access through assembly language is quite low level and would be used to program device-dependent functions. This book is concerned with programs that utilize the flexibility provided by the ST; therefore, the C function calls are used.

The GEM VDI can be thought of as one routine. This routine knows how to represent *all* the graphic images produced by GEM; it also performs a wide variety of different graphics functions. To tell the

routine what you want done, use an operation code (opcode, for short), which is then passed to the routine.

Besides the opcode, most functions require a set of parameters that define the operation. To draw a circle, for example, you need to tell the machine where to put the center point and how long to make the radius. You supply these variables using x and y coordinates to plot the center point and the length of the radius along the x axis. The parameters are passed to the VDI function through a set of global arrays (values assigned throughout the system). Then the VDI plots your graphics display.

There are five global arrays of integers used by the VDI functions that must be defined by your C program. These arrays include control values, input and output integers, and input and output vertices (see Table 1-1). The control values contain a variety of information such as the opcode, the number of elements contained in the other four arrays, and the output device number. The input and output integer arrays are used to pass integer values between the VDI routine and the program that uses them. The input and output vertices arrays hold the points to be plotted by the VDI to create the desired graphics display.

Table 1-1: Global Parameter Arrays Used in the VDI

Variable Name	*Description*	*Number of Elements*
contrl	Control values	12
intin	Input integers	128
ptsin	Input vertices	128
intout	Output integers	128
ptsout	Output vertices	128

Because the arrays are linear (one-dimensional) and a vertex (a point) requires two coordinates, GEM has adopted a standard format for storing points in the array. For each point, the x coordinate is given first followed by its corresponding y coordinate. Since all arrays in C start with element zero (0), the x and y coordinates of the first input point are placed in elements intin[0] and intin[1], respectively. Those of the second input point are placed in elements intin[2] and intin[3].

It is very time-consuming and inefficient to have to fill in every parameter array and call each routine by number. However, there is no need to do this. C provides a set of C "bindings" that provide easy access to GEM as well as Atari functions. Each binding is merely a C

function call interface that allows you to call up an entire VDI operation without filling in the opcode and array elements. Suppose you want to draw a circle. The function call for a circle in C is named **v_circle( )**. You would only need to supply the parameters for the x and y coordinates of the center and the length of the radius. The function fills in the appropriate elements in the control and point arrays for you. The Megamax compiler defines many of the function names in header files used in the programs in this book (see Appendix B, Header Files).

The primary reason for mentioning these arrays is that they must be defined somewhere in your program. Chapter 3 discusses precisely how to do this. Also, if you plan to do any assembly language programming with GEM, these parameter arrays will be the primary form of communication between your program and GEM. Throughout the rest of this book, parameter arrays are merely mentioned for completeness. Little emphasis is given to them unless required.

The AES operates in a manner similar to the VDI. There are seven parameter arrays for the AES: global values, parameter blocks, control values, input and output integers, and input and output addresses (see Table 1-2). Since these arrays are already defined in the AES libraries, you don't need to define them within your program like the VDI arrays. In general, you don't need to access the AES arrays for your function calls. The size of these arrays depends on the implementation of the AES. Since you don't need to declare the arrays, you don't need to know their size. However, they are global variables. If you want to access them, simply declare them as external variables within your program.

Table 1-2: Global Parameter Arrays Used in the AEB

Variable Name	*Description*
control	Control values
global	Current status values
int_in	Input integers
int_out	Output integers
addr_in	Input addresses
addr_out	Output addresses

Writing a C Program

There are many good compilers available for the C language on the Atari ST; we use the Megamax. Whichever compiler you use, you should familiarize yourself with how to create simple programs with

a compiler before you attempt the programs in this book. The Megamax compiler for the Atari ST comes with an application program called SHELL. PRG. This application provides a more convenient programming environment than the desktop.

When writing a C program, you first need an editor capable of writing the program source files (the code that you write) in a nonformatted file. Source files should not contain special word processing characteristics but only plain text. You may name your source file anything you wish within normal file-naming parameters as long as it has the letter "C" as its file extension (for example, program.c, source.c, or myfile.c.

The next thing you need is a compiler. Since different compilers operate differently, refer to the compiler manual to determine proper operating procedures. With the Megamax compiler, you first initiate the compiler program and then enter the source file's name. If you are using the shell, you execute the compiler and select the source file from the file selection box. The source file is processed (compiled) into an object file, which is an intermediate form of your program. The object file has the same name as your source file, but with the letter "O" as its extension such as program.o, source.o, or myfile.o.

Finally, you need a linker. A linker, as its name implies, links files together. In C, you can create separate object files that contain different discrete functions of your program. A linker combines these object files into one program file. Also called a binder, a linker binds your program's references to the operating system with a code that interfaces with the system. As this code is supplied by the compiler manufacturer, you need not be overly concerned with it. For the Megamax compiler, you initiate the linker and enter in all the object files to be linked. Since the order may be important, check the compiler manual. If you are using the Megamax shell, execute the linker, move the appropriate object files to the link list, and select "OK" to begin linking. All the programs in this book use only one source file and one object file; the interface code is automatically included by the Megamax linker.

One other useful program is a resource construction program. This program creates resource files for application programs. Resource files contain data for dialog boxes, alert boxes, and menus used by the program. Resources and the resource construction program are discussed later with the AES.

Take time now to read the compiler manual. Write and compile some very simple C programs. Once you know how your compiler works, writing more elaborate programs becomes easier.

CHAPTER TWO

Picture This—An Introduction to Computer Graphics

"A picture is worth a thousand words." If this were not the case, computer graphics might never have become a reality. However, the adage is true, and computer graphics has become an industry unto itself. This chapter shows how computer graphics are created and, more specifically, how the Atari ST generates graphics displays. This chapter also introduces various kinds of input devices commonly used with interactive graphics systems and explains how these devices work. All of this is a preface to the concepts and terminology used by GEM to implement a graphic interface that is easy for both the computer operator and programmer to use.

Background

The realm of computer graphics covers a wide variety of devices and techniques. Just explaining the various graphic devices now available would take a book by itself. The discussion of computer graphics in this chapter limits itself to the Atari ST and its GEM environment.

The Pixel

Look closely at the Atari ST screen after you turn it on. From a distance you can see the GEM desktop, but up close you can see dots. Each dot is called a picture element or *pixel*. Your entire screen is really a grid or matrix of these pixels. Pixels are more visible on a color monitor than on a monochrome monitor because the pixels are

closer together on a monochrome monitor. The closer together the pixels are, the better the quality of the picture created. This difference in pixel placement and picture quality is called *resolution*. An example of very low resolution would be 10 dots per inch. In this mode, the output device would have only 10 dots to draw a line one inch long. Naturally, a picture that uses 10 dots for a line would be less well defined than one that uses 50 dots for the same line. Most dot-matrix printers provide a resolution of 70 to 150 dots per inch. Laser printers, known for their excellent resolution, can have as many as 300 dots per inch.

A display on a monochrome monitor is created using the two different colors a pixel can display. These colors are usually black and white, black and amber, or black and green. Every pixel on the screen can display either color, depending on whether it is turned on or off. On the Atari ST, an "on" pixel shows black and an "off" pixel shows white. This "on/off" terminology probably reminds you of the yes/no configuration for bits in computer memory. In fact, the two are closely related. Creating the range of hues available on a color monitor is a little more complex, but it builds on the techniques used for monochrome monitors. Color monitors are covered later in this chapter.

Display Technology

The first step toward understanding how screen displays (and other types of displays) work is to see how a computer monitor physically creates the light for a white pixel (the "off" designation in the case of the Atari). The inside of the monitor's glass screen is coated with phosphor, a substance that glows when an electron (an electric particle) hits it. Screen monitors are equipped with electron guns which project electrons onto the screen. When an electron hits a point on the screen, a white dot shows. The phosphor glows for only a fraction of a second; to keep a dot glowing continuously, the gun creates a beam of electrons. There are, of course, many dots on a screen that must be lit to create a single display. Since it would be impractical to have an electron gun corresponding to every pixel on your screen, the gun's beam scans across the screen line by line. When the beam reaches the bottom corner, it moves back to the opposite corner on top and starts the process all over again. This happens very quickly; scanning the entire screen takes about one sixtieth of a second.

As the beam scans the screen, it turns on for each dot that should be glowing and off for each dot that should be dark. The question now arises, "How does the computer know which pixels are on and which are off, and how does it remember from one pass to the next?" It's rather simple, really. The computer uses its memory. Actually, it uses a portion of its memory to create a map of each pixel's position

on the screen. For each bit in memory set to 1, the corresponding pixel is considered in the "on" state. For each bit set to 0, the pixel is considered off. As the electron beam scans the screen, the display hardware reads the memory and controls the beam output. This type of display architecture is called a *bit-mapped display* because the individual pixels are mapped to bits in memory.

Using a Bit Map

The portion of memory labeled the bit map is reserved for use by the display processor. The bit map may actually be located anywhere in memory as long as the display processor knows where the bit map begins. To facilitate this, the base address of the bit map must be loaded into the display processor. The Atari ST XBIOS provides routines to load the display processor with the bit map's base address. These functions are covered later.

To use the display bit map to create graphics, place a bit within the map. The corresponding pixel will immediately be plotted on the screen. This is a very direct approach to generating graphics, and it is not usually used. The advantages of this approach are that it is a very fast access method and it gives the programmer direct control over what is displayed on the screen. The disadvantages include the fact that the programmer is responsible for much additional processing overhead and that the bit map display is only practical for the screen. This method is completely device-dependent.

Making Pictures

Instead of directly accessing the bit map, a set of commands can be used to draw *graphic objects* such as a point, a line, or a rectangle. These routines do not usually include very complex objects like circles, ellipses, or houses. Instead, the basic objects are used as foundations to build more complex images. These images are then implemented in the output hardware or through some extremely efficient software. If this were not the case, the job of drawing more complex objects would be terribly slow and extremely tedious because you would need to place every dot in its exact place.

One set of graphics routines available on the Atari ST is called the Line A Handler. Atari engineers have utilized the Motorola 68000 microprocessor's unimplemented instructions to provide "quick-and-dirty" access to assembler-level graphics routines. The Atari ST documentation states that "if an application only needs a few primitive graphics functions (and wants maximum performance), then

Line 'A' is sufficient (and optimal)." In case you're wondering about the name Line A, it comes from the fact that the 68000's unimplemented instructions all begin with the hexadecimal digit "A". The GEM VDI provides most of the functions found in the Line A Handler. In fact, the VDI uses Line A for many of its routines and VDI routines are easily accessible using most programming languages. Because direct access to the Line A routines is available only through assembly language, we do not discuss the Line A Handler beyond what has been presented here. If you are interested in directly accessing the Line A routines, you need a book on 68000 assembly programming and the Atari documentation for the Line A routines.

GEM VDI provides a programmer with a wide variety of high-level graphics operations. For instance, you can draw circles, ellipses, rectangles, polygons, and arcs. You can also fill any closed polygon you draw, not just with black dots but with any type of pattern you specify. You can even draw lines and text in various patterns simply by specifying the pattern you want and using the proper VDI functions.

The bit map and the VDI are two separate entities. The VDI uses a bit map to generate a graphics display. The VDI provides the interface between the program and the bit map. The VDI also provides the interface from the bit map to graphics devices that may or may not use bit maps. For example, a plotter or dot-matrix printer does not have the same capabilities as a video monitor. A plotter or printer cannot place a dot on paper and then remove it. Once a dot is placed on the page, it is there to stay. For these devices, you would use the VDI to draw an image and then tell the VDI to output the image to the device. Unlike the bit map for the Atari ST screen, you do not have access to the internal VDI representation for other devices.

Uses of a Bit Map

The concept of a bit map is a quite powerful one. First consider that a bit map can be any size to accommodate various tasks. If the bit map is being used as the storage representation for a monitor, the map will contain enough bits to represent each pixel. However, there are also smaller bit maps available from the Atari system. Suppose you had a small bit map that was 16-by-16 bits in size. What would you use such a bit map for? One possibility is illustrated in Figure 2-1. In this bit map, each 1 bit is a black pixel and each 0 bit is a white pixel. (The Atari inverts the usual 1-on/0-off representation because it uses black images on a white background.) The configuration in the illustration forms a checkerboard pattern (Figure 2-1). If

```
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
```

Figure 2-1 16-by-16 Bit Map for Checkerboard Pattern

you store a small bit map like this in memory, you can use it to fill a rectangle or other closed polygon by copying it over and over until the shape is filled. Transferring data from one place in memory (the pattern bit map) to another place in memory (the screen bit map) is much faster than calculating how to fill in the rectangle with a complex pattern. Therefore, one use for small bit maps is pattern definition for area fill.

Now look at Figure 2-2 where dots have been used in place of zeros to make the image more visible. Does the image look familiar? This bit map is a representation of the icon used by the desktop to represent a data file. A 32-by-32 bit map is used to store icon images (Figure 2-2). Instead of redrawing the icon each time it is needed, a program can simply copy the icon's bit image into the screen's bit map.

A third use for bit maps is text appearance. Every character on the keyboard can be stored in an array of bits called a character cell. If you are familiar with word processing or typesetting, you are probably aware that text has some characteristics unique to its representation such as fonts or typefaces, type styles, and type size. The terms *font* and *typeface* are synonymous and refer to the look of the letters. Some examples of fonts or typefaces are *Roman*, *Script*, and *Futura*. The *type style* refers to modifications on the typeface. These include options such as *italicization*, underlining, **shadowing**, and **boldface**. The *type size*, as the term implies, indicates how large the letters are. GEM VDI allows you to set the type size in coordinate units (using pixel-to-pixel distance such as 20 units high) or in points (using the printer's measurement of 1/72 of an inch).

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . . . . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . . . . . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 1 1 . . . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . . . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 1 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . . . . . . . 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 1 1 1 1 1 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 . . . . 1 1 . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 . . . 1 1 . . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 . . 1 1 . . . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 . 1 1 . . . . 1 . 1 . . . . .
1 . . . . . . . . . . . . . . . 1 1 1 . . . . . 1 . 1 . . . . .
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 . . . . . . 1 . 1 . . . . .
.
.
.
```

Figure 2-2 32-by-32 Bit Map for Data File Icon

For each character in a font, which generally includes all characters on the keyboard, a character cell needs to be defined. Each type style must also be defined in individual character cells for every font that will use that style. In addition, for each different type size a new set of character cells for each font and style needs to be created. Although software algorithms exist to enlarge typefaces and change type styles given a base font, the characters never look as good as those created from character cells. This means that an ambitious person needs to program the cells for each font, style, and size needed.

There are a number of variables that can be set for each character cell. These have to do with the width and height of the individual characters and the cells that contain them. Each character is assigned a cell which determines how much space a character occupies

and how much space is left around it (between it and other characters). Vertically, each cell is divided by six lines. These lines that start at the top and go down are called the top line, the ascent line, the half line, the base line, the descent line, and the bottom line (see Figure 2-3). Bottom and top lines define the vertical limits of a single cell, or the *cell height*. The bottom line of one cell in a block of text is usually defined as the top line of the cell directly below it. The characters in most cases do not use the total available cell height. Rather, they extend up to the ascent line if they are full-sized letters (upper-case or tall lower-case letters) down to the descent line if they must extend below the base line (as with lower-case letters such as "g" and "j"). The half line defines the maximum height for most lower-case letters and small characters, and the *base line* defines the bottom of a character.

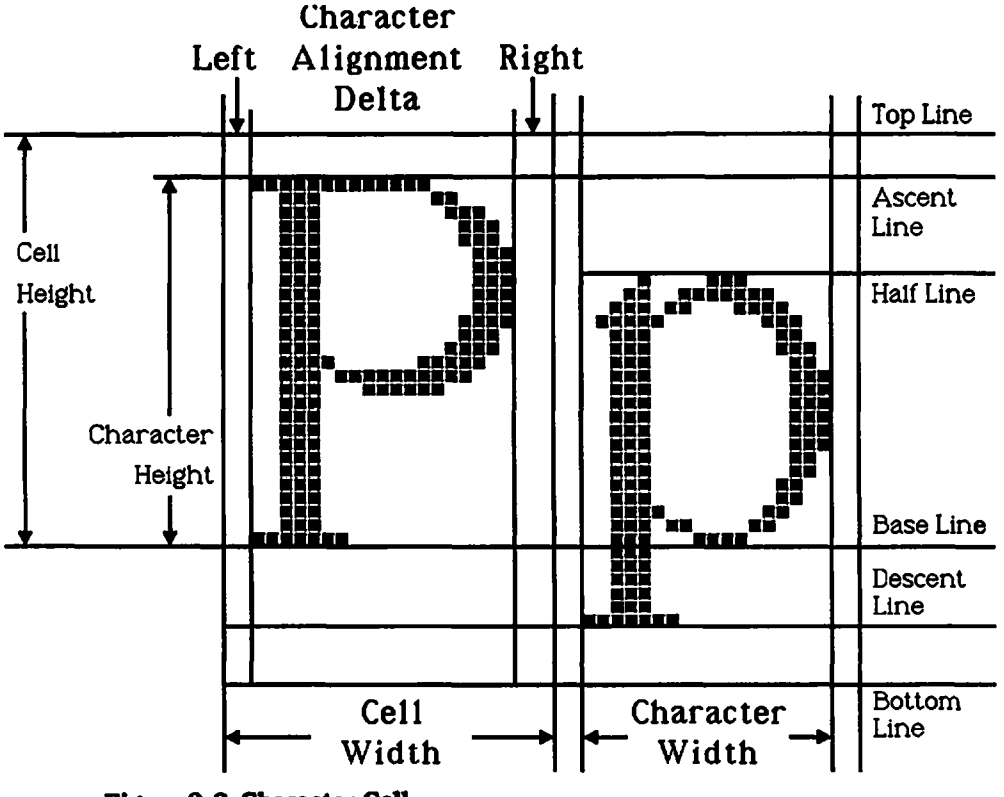


Figure 2-3 Character Cell

Defining the width of characters and character cells is somewhat similar to defining the height where the character width is usually less than the cell width. This allows for space between characters. The

actual commands used to define typefaces, type styles, and type sizes are discussed later in this book. For now, it is important to realize that each character cell is a small bit map that can be called up instantly from memory for use in a display.

Logic Operators

Basic logic operations performed in memory are used to change pixel values. Logic operations act on the values *true* and *false*. In computers, these values must be translated into the binary values 1 and 0. For the purposes of this discussion, let *true* have the value of 1 and *false* have the value of 0. Logic operators work in much the same way as mathematical operators. The operator takes two operands (values), performs an operation on them, and produces a result. The four logic operators commonly used with computers are OR, AND, XOR, and NOT.

The first operator, *OR*, tests if either of its two operands is true (equal to 1). If that condition is met, the operation gives a result of *true*. If both operands are false (equal to 0), the operation gives a result of *false*. Table 2-1 is the logic table for OR. The x and y columns refer to the two operands. The result is listed in the x OR y column.

Table 2-1: Logic Table for OR

x	*y*	*x OR y*
T	T	T
T	F	T
F	T	T
F	F	F

The next operator, *AND*, tests if both of two operands are true (equal to 1). If this condition is met, the operation gives a result of *true*. However, if either operand has a value of false (equal to 0), the entire operation gives a result of *false*. Table 2-2 shows the logic table for AND.

Table 2-2: Logic Table for AND

x	*y*	*x AND y*
T	T	T
T	F	F
F	T	F
F	F	F

The third operator, XOR, is quite useful when dealing with graphics. The XOR operator produces a *true* result if the two operands have different values. In other words, if x and y are different values, the operation tests true. If they are the same value, the operation tests false (see Table 2-3). The interesting aspect about XOR is that it acts like a toggle. If you have a value and XOR it with some arbitrary value, you get some result. If you XOR this result with your original value, you get the original arbitrary value back. For example, if you have the value 1 and XOR it with 0, the result will be 1. If you XOR this result of 1 with your original value of 1, you get back 0. An example of how this is useful is given soon.

Table 2-3: Logic Table for XOR

x	*y*	*x XOR y*
T	T	F
T	F	T
F	T	T
F	F	F

The last operator, *NOT*, is a unary operator. This means that it only uses one operand. NOT inverts the value of the given operand so that NOT *true* yields *false* and NOT *false* gives a *true* result.

The logical operators are essential to the creation of graphics displays. They are used, in some aspect, in every graphics operation.

Writing Modes

In the initial discussion of bit maps and the VDI, it was stated that certain writing modes available with the bit maps would have minimal effect on a program output to the screen. This section explains the four writing modes available from the Atari for use with bit map displays. Writing modes are methods of taking bit maps from memory and plotting them onto the bit map for the screen display. The four modes available from the Atari are *replace*, *transparent*, *XOR*, and *reverse transparent*. Like the examples given above, the writing modes use two bit maps: a source bit map held in memory and a target bit map that is normally the screen.

The simplest of the writing modes to understand is replacement. In this mode, each bit in the source map has a corresponding bit in the target map. The bits are simply transferred from the source to the

target map. Thus, if you have a 0 bit in the first position of the source map, the first position in the target map also becomes a 0.

For the remaining modes, the source bit map and the target bit map go through a logic operation with the result placed in the target bit map. For example, in transparent mode the pixels set to 1 in the source bit map are copied to the corresponding bit in the target bit map. Those pixels that are 0 in the source bit map have no effect on the target bit map.

In XOR mode, each bit in the source map and the corresponding pixel in the target bit map go through the logical operation XOR explained above. If you have a 1 bit in the source map and a 1 bit in the target bit map, the pixel in target bit map is set to 0. If you have a 1 in the source bit map and a 0 in the target bit map, the resulting pixel in the target bit map is a 1.

The final writing mode available from the Atari ST is reverse transparent mode. This mode, as its name suggests, works in a manner exactly opposite from the transparent mode. In other words, when a bit has the value 0 in the source bit map, the corresponding pixel in the target bit map is set to 1. The descriptions given here for bit maps and writing modes apply primarily to monochrome monitors. The use of these writing modes is explored shortly.

Bit Map Representation

The use of bit maps for icons and text representation assumes that the source and target bit maps are the same size. In reality, the VDI and most graphics applications use this function by having two *different* sizes of bit maps. For example, an icon bit map that consists of a 32-by-32 bit square is not the same size as a monochrome screen, which consists of 640-by-400 bits. The bit map is memory for the icon is used as a mask on top of the bit map for the screen. By locating the upper left-hand corner pixel on the screen, the bit map of the icon can be overlayed onto the bit map of the screen.

This sounds quite simple, but the actual representation of a bit map is a little more complex. Memory itself is linear: one bit follows another bit and thus creates bytes (eight consecutive bits), words, and so on. A bit map is more of an array with height as well as width. As in memory, each bit in the bit map is contained in a byte. These bytes are arranged in lines along the map. When the computer reaches the end of a line in the bit map, as it is recording the map in memory, it simply continues to the next line in the map. All this is handled by the VDI and is discussed in more detail later in the book.

Output Devices

Obviously, not all output devices are the same. Differences between a screen and a printer have already been demonstrated through the single example of the use of the bit map. Output devices also vary within the same category. For instance, a color monitor has very different needs and capabilities from a monochrome monitor. Even a monochrome monitor can come in a variety of display modes: bit map display, vector display, and storage tube display. Devices that produce so-called hard copy also differ widely in their capabilities. Plotters, laser printers, and dot-matrix printers all use different modes to produce output, and there is a great deal of variety within each of these different types. Plotters can use a bit map, a pen, or a drum to place an image on paper. In some cases when preparing a graphics operation, the programmer needs to consider how the image is produced. For example, a pen plotter that uses a pen to draw lines does not use a bit map to coordinate the lines. A bit-mapped screen, however, does use a bit map. A dot-matrix printer can translate a bit map from memory to output on paper; however, it must go through some changes because a dot-matrix printer prints seven or more dots vertically at one time, while the bit map is arranged horizontally. Although GEM does quite well in eliminating these problems, some consideration still has to be given to output, especially for programs that are more complex.

There are some advantages and disadvantages to the capabilities of the different devices. The bit-mapped screen's great advantage comes from the writing modes. These make it possible for a program to erase or show any pixel in any position on the screen virtually at any time. This capability allows a programmer to create representational graphics such as those used in design and animation. Disadvantages include the fact that the resolution of the screen is limited to the number of pixels the screen can display. For example, if you connected the Atari to a television set, the characters are far less readable than those on the monochrome monitor.

A vector display creates output with an electron gun, just like the bit-mapped display. However, with the vector display the gun draws lines rather than points. To obtain a line, you specify the start point and endpoint. This creates superb lines that can be used together to produce high quality graphics. The disadvantage is the amount of time it takes to create the lines. The overhead with this type of graphics display is such that the number of lines you can include for any one image is limited.

Storage tube displays work in a manner almost identical to vector displays. The exception is that once a line is drawn to the screen, it remains visible until the entire screen is erased. Therefore, if you

draw something, the only way to get rid of it is to get rid of your entire display. This has some obvious disadvantages in terms of the flexibility of the display. If you want to change a single line, you have to erase the display and redraw it without that line. However, if you are ready for final output, this type of device can provide extremely fine images.

Plotters have problems similar to those found with the vector and storage tube displays. A pen plotter acts like a vector display in that you put the pen down where you want to start drawing a line and lift it at the end. While pen plotters have very good resolution, they require much time overhead to create images. Also, once a line is drawn, the only way to get rid of it is to use a new sheet of paper.

Printer types can basically be split into two groups: impact and nonimpact printers. The impact printers include letter-quality (such as a daisy wheel) and dot-matrix printers. Nonimpact printers include ink-jet; thermal, and laser printers. Printers that produce only letter fonts, such as daisy wheels, are almost useless in terms of graphics output, unless you're interested in making pictures of the Mona Lisa using Xs. Most people using this book have a dot-matrix printer. Dot-matrix printers use a set of seven or more pins to place patterns of dots that create graphics displays on paper. Laser printers work on much the same principle as a bit-mapped display. The image is created by tiny beams of light that are projected in patterns as defined in a bit map; these form the graphics display.

Device Coordinates

When you graph an image on a piece of graph paper, you place points on a grid according to their coordinates. On a computer you must also specify where you want your points placed. This is done on a two-dimensional surface called a *drawing plane*. The drawing plane is simply a concept that provides an imaginary surface to draw on. Just as with graph paper, the drawing plane can have any type of coordinate system you choose. For example, you can use the standard Cartesian coordinate system. In this system, the point (0,0) is placed in the center; positive values increase as you go up and to the right; and negative values increase as you go down and to the left. You can choose a system where the point (0,0) is in the lower left corner and use only positive coordinates. You can also place the point (0,0) in the upper left corner and have positive values increase as you go down and to the right. This last coordinate system is the one most commonly found on computer displays, particularly bit-mapped displays. The reason for this is that the electron gun scans from top to bottom

and from left to right; thus the bit map should follow this order for efficiency.

The device coordinates you use are dependent on how high the resolution is on the output device you are using. Most of the time, device coordinates are represented with whole numbers (for example, 1, 2, or 3). These numbers specify some position on the page or screen, and the range varies according to the device you are writing to (for example, 0–400 or 0–4000). Every device has its own set of coordinates, most of which start at zero (0). Some devices, such as screens and plotters, allow you to access any point at any time. Other devices are able to move in only one direction; for example, since most printers can only move down the page, your image must be drawn line by line. Thus, on a printer your device coordinates consist of the line number on a page and the dot number on a line. On a screen or plotter the coordinates consist of the actual horizontal and vertical positions. These coordinates will, of course, vary from output device to output device.

Monochrome Versus Color Screens

When you are dealing with monochrome graphics, you can either have black or white; this corresponds quite nicely to the yes/no or 0/1 logic computers use. When you use color graphics, you encounter a whole new range of problems.

All colors can be created using the three primary colors of light: blue, green, and magenta (red). The presence of all three colors at once gives you white and the absence of any color gives you black. If you add green and magenta, you get amber (yellow/orange). If you add green and blue, you get aquamarine. If you add blue and magenta, you get purple. Other shadings and nuances are achieved by varying the intensities of the light colors being mixed.

The technical aspects of color representation include the use of three electron guns instead of the single gun used in monochrome display. Each gun activates a different color of phosphor: magenta, green, or blue. Combining the output from the three guns creates color displays on your color monitor. To control which colors are activated, you must have at least three bits where one is for magenta, the second is for blue, and the third is for green. You can then have eight different color representations ranging from black to white simply by combining these three bits in different combinations. To allow an even greater range of color, a fourth bit can be added to adjust the

intensity of the color. This gives you eight low-intensity and eight high-intensity colors.

Another way of handling color is to allow the individual colors to vary their intensity. To do this, you use one byte in memory to determine the intensity of a particular gun for a particular pixel. Thus, if you have an 8-bit byte, you can have 256 intensities. If you have three bytes of magenta, blue, and green each with 256 intensities, you have a range of over 16 million different color representations per pixel. However, to represent this range you must have three bytes of memory per pixel. For the Atari ST, this means that in medium resolution (320 by 400 pixels), you must have over 350,000 bytes (350KB) just to take care of the color representation. That's not a very efficient or logical use of memory.

To reduce this memory requirement, the Atari uses *color planes*. In essence, each of these planes is the bit map for a color. In low resolution mode there are four planes; in medium resolution there are two planes; and in high resolution (used only in monochrome) there is only one plane because the only colors represented are black and white (0 or 1). In the color modes, the computer combines corresponding pixels in each plane to obtain a binary number which is then mapped to a color. The Atari is capable of displaying 512 different colors. However, because in low resolution the maximum number of bits available is four (one for each of the four planes), the maximum number of colors available for display at any one time is sixteen. There are ways around this limitation that use advanced graphic techniques. For most uses, though, sixteen colors should be enough.

If you go into your control panel accessory in the desktop, you see three slide bars on the left. On a color monitor each bar represents one of the three colors: magenta, blue, or green. By moving the slide bar to the top, you get maximum intensity. Moving the slide bar to the bottom gets you zero intensity in the particular color. There are eight different intensities for each color. If you multiply the eight red intensities by the eight blue intensities by the eight green intensities, you get a total of 512 color combinations available. On the bottom of the control panel are 16 squares. On a monochrome monitor there is one white and 15 black squares. On a color monitor each square shows one of the available colors. This is your color palette, which the system uses by default. There is a table in memory that maps the values 0 through 15 (the 16 colors in your palette) to 16 of the 512 available colors. You can have any combination of colors you desire, and you always have 16 colors available.

The use of color in a program requires a full understanding of the color palette and its construction. Further discussion of color graphics is in Chapter 6.

Input Devices

Just as there are many different types of output devices, so there are a variety of input devices available for use with computers. These can be generally broken down into four basic categories: keyboards, locators, valuators, and buttons. Probably the most familiar is the *keyboard*, a device for entering text. A *locator* is a device for pointing to an object or line of text shown on a screen. The most common locator is a cursor, which can be controlled by various physical devices. The third input device, a *valuator*, is a logical device that provides a range of numeric inputs, for instance, a dial or a slide bar. Finally, a *button* is a device for selecting from a number of different displayed options. These are the logical, or conceptual, input devices. The physical implementations of these are explained below.

The function of the keyboard as an input device has come to be taken for granted. It is a layout of letters and specialized characters produced on the screen by pressing the character's corresponding key. It is almost identical to the function of a typewriter's keyboard. Another way a keyboard can be implemented is to represent the keyboard on the screen and use a locator to identify the key to be entered.

Locators are relatively new input devices for personal computers. The most widely used physical locator is the mouse, which usually consists of a small box containing a roller ball. The mouse controls cursor movement on the screen. Other locators include devices such as light pens, which physically point at an object on the screen. Joysticks are a third type of locator; like a mouse, they move the cursor on the screen.

The valuator is usually implemented as a dial or a slider. The user can change the position of the valuator and cause a new value to be produced. This value can represent anything from sound volume to one coordinate of the cursor's position.

The last logical input device, the button, is usually just a button. A button is often found in combination with locators. For instance, in addition to the tracking ball the mouse box usually has at least one button, which can be used to select information on the screen. Joysticks also usually have buttons that provide some type of input to the screen, depending on the software being implemented. Function keys are yet another type of button. They are most often included on the keyboard but may be physically located away from the usual character set. They are identified by the letter F followed by a number, usually 1 through 10. These buttons are quite special, as they change their function depending upon what program you are running and where in that program you are. They are, therefore, referred to as programmable function keys.

Implementing Logical Devices

On the Atari ST, a mouse is used as the locator. Valuators can be taken from the keyboard or from an object on the screen (for example, the slide bars on the windows). Buttons on the Atari include the function keys at the top of the keyboard, the buttons on the mouse, and particular keys on the keyboard (depending on how they have been programmed for application). The keyboard, of course, acts like a normal keyboard unless you program it otherwise.

The Ideal Graphics Device

A concept much referred to in computer graphics is the *ideal graphic device*. This is not an actual physical output device. Rather, it is a concept that provides the user with anything required at the time the user calls on a graphics routine. This ideal graphics device provides a bridge between the actual output device and the program.

The ideal graphics device uses what is called a normalized device coordinate (NDC). This means that the coordinates on the ideal graphics device are consistent. Most ideal device systems have a coordinate system where the x values range between 0 and 1 and the y values range between 0 and 1. This is a true set of normalized coordinates. In practice, the actual range is irrelevant as long as it is consistent. GEM uses a range of 0 to 32,767 for both the x and y coordinates. The origin of this system varies with the implementation of the ideal device. For GEM the origin coordinate (0,0) is located in the lower left corner. Another feature of NDC is that the distance between pixels is the same in both directions. This means that the grid is made up of squares such as those on graph paper. Also, the coordinate values increase from left to right and bottom to top.

The alternative to normalized device coordinates is actual device coordinates—called raster coordinates (RC) in GEM. There are several differences between NDC and RC. First, the RC origin may vary. As mentioned earlier, most computer displays have the origin in the upper left corner. Another difference is that the range of coordinate values depends on the resolution of the device. Also, the direction in which coordinate values increase can vary. Finally, the distance between pixels may not be the same in both directions. The problem with this attribute is that a circle on one device may appear as an ellipse on another device. When you work in raster coordinates, your program becomes device-specific.

Ideally, you want your program to remain as device-independent as

possible. Therefore, you want to use NDC for your graphic output. GEM allows you to use NDC for your programs. Your program draws its images using NDC, and GEM uses its device drivers to output the image using the raster coordinates. The device driver takes into consideration all of the device's attributes such as resolution and pixel distance. The device driver converts the image so that if you draw a circle and output it to a screen, plotter, printer, or other output device, it looks like a circle. It has the same shape and size (relative to the other objects in the image) on each device.

The GEM Workstation

The *workstation* is GEM's implementation of the ideal graphics device. It contains all capabilities of the ideal graphics device such as using NDC. The GEM workstation contains a large number of graphic attributes that you can set. These attributes tell GEM how objects such as lines should be drawn. For example, a line can be drawn solid, dotted, dashed, dot-dashed, or in some other fashion. Once you set the particular attribute, you don't need to worry about it. If you want to change it, simply set it to a new value. The graphics attributes provide you with a wide range of tools to help you create spectacular graphic images.

Other workstation attributes include fill settings, line settings, text settings, writing modes, and clipping. Fill settings correspond to commands that cause areas of the screen to be filled, like drawing a filled shape. Fill settings cover the color to be used, type of fill (hollow, solid, pattern, hatch, or user-defined), and pattern or hatch selection. Line settings include line width, color, type (as mentioned above), and end point styles. Text settings include font selection, size, special effects, color, and rotation. The writing modes (replace, transparent, XOR, and reverse transparent) have already been introduced. All of these attributes are defined for the workstation. Any VDI output function uses some combination of these attributes when it produces its image. The current attribute setting may be changed at any time. Any output following the change uses the new attribute value. Table 2-4 shows the default values for some of the workstation attributes. All these attributes are explained and demonstrated in greater detail in Chapter 4.

The last attribute listed in Table 2-4, clipping, is actually a function done by GEM. The program can specify a rectangle on the workstation which outlines the area to be visible. Any point a program tries to draw outside of the rectangle is not drawn. Lines drawn from within

Table 2-4: GEM Workstation Attribute Defaults

Attribute	*Default Setting*
Character height	Nominal character height
Character base line rotation	0 degrees rotation
Text alignment	Left base line
Text style	Normal intensity
Line width	Nominal line width
Marker height	Nominal marker height
Poly-line end styles	Squared
Writing mode	Replace
Input mode	Attempts to use all types of input
Fill area perimeter visibility	Edges will be drawn
Line style	Solid
Fill pattern	Solid
Cursor	Hidden
Clipping	Disabled

the rectangle to the outside show only that portion of the line that falls within the rectangle. Clipping is also demonstrated in Chapter 4.

You now have a basic understanding of the GEM VDI and its capabilities. All output for the GEM VDI is done on a workstation. The next chapter develops the basic routines needed to get an application running on the Atari St.

CHAPTER THREE

Preparing to Use GEM

GEM, as its name implies, is an environment manager. From the programmer's perspective, it provides the tools that help produce graphics-based programs. In addition to the sophisticated graphic images that can be produced, GEM provides independence from output devices and portability to many host computers.

All these features do not come free. GEM places the small but important restriction that you follow certain sequences of procedures while your program is executing. These procedures tell GEM what your program wants done, how it is to be done, and what attributes are to be used. The most basic program would perform the following sequence of operations:

Initialize the application for GEM;
Locate the physical input/output device for the system;
Obtain an input/output device for the program;
Do the application's procedures;
Release the program's input/output device;
Terminate the application from GEM.

Initializing the application for GEM performs two steps. First, it identifies the program to GEM, and second, it allocates memory for the program to use. Identifying the program to GEM allows GEM to track input to and output from the program. GEM is designed to be a multitasking environment, which means that more than one application may be active at any given time. This condition is best

demonstrated on the Atari ST by having a program running (an editor, for example) and accessing one of the desk accessories. When you request the desk accessory, it becomes available on the desktop. The editor is still active and you can switch from the accessory to the editor and back again. GEM must know which application—the editor or the accessory—is currently running so that when the user provides input, GEM can direct the input to the proper program.

When a program is running, it needs memory to perform its processing. The second part of initialization tells GEM to allocate memory for this program. If your program does not perform this initialization step, you will undoubtedly interfere with GEM's operation. This can result in improper program execution, loss of a device such as the disk drive or mouse, or a system crash (also known as a *bomb*).

The next two steps your program must perform locate the input/output devices used by the system and your program. A true Atari ST application causes all output to be written onto the program's output device or to a window. If you are just writing a quick little program that uses only the C/UNIX-compatible text output commands such as printf( ), you won't need to access these devices. You also won't be able to access any of GEM's graphic and text output capabilities.

The system and program input/output device is implemented using the workstation concept introduced in Chapter 2. The use of a workstation and the reason a system and program workstation is required are discussed in more detail in the next section.

The next step in your program is for it to do its processing. This can be anything you program your application to do. Once your program has finished its processing, it must return its resources to GEM. The first resource the program returns is the workstation. Returning the workstation to GEM releases the memory allocated to it and makes it available to other programs.

The final step is telling GEM that the program is ready to terminate. This allows GEM to get the second resource used by the program—its memory workspace. GEM terminates the program and returns its memory to the available pool. The program termination also removes the program from GEM's list of active processes so that it no longer receives input from the system.

These six steps provide the very basic outline of a GEM application. The next few chapters focus strictly on the VDI. This means that the programs do not use menus, windows, dialog boxes, or mouse input. When the AES is introduced, a few more steps need to be added. In the meantime, the above list of steps is the basis for the application skeleton developed in the remainder of this chapter.

Workstation Usage

Chapter 2 introduced the concept of a workstation. A workstation is a mechanism through which a program can develop a graphic output environment. The workstation keeps track of the attributes currently being used for graphic output. However, this is not the only information the workstation uses. The workstation also identifies the location of the graphic device, its output capabilities, its input capabilities, and its physical attributes. For a full list of workstation data, see the VDI functions **v_opnwk( )** and **vq_extnd( )** in Appendix A.

When your application begins its execution, it requests from GEM the workstation used by the system. This workstation corresponds to the video display, keyboard, and mouse (collectively known as the console) of the Atari ST. GEM identifies the workstation by returning a number called a *handle.* A handle is simply a data item (in this case, a number) used to uniquely identify some object. For example, the C function fopen() returns a pointer to a FILE object. The pointer is a handle. The C function open() returns an integer identifier to the file. This integer is also considered a handle. Once you know the handle of a workstation, you always use that handle when you want to use that workstation.

After you obtain the handle to the system workstation, you need to create a workstation for use by your program. This is done through a process called opening a workstation. When your program opens its workstation, you often want to use the screen and the keyboard as well. Unfortunately, GEM is already using the console for the system workstation. Fortunately, GEM provides a way around this problem.

GEM uses two types of workstations: a physical workstation and a virtual workstation. A physical workstation is a workstation attached to a physical device such as a console, printer, or plotter. Each device has a unique device identification number and a corresponding device driver (see Table 3-1). The devices available to the system, their ID numbers, and their device drivers are listed in a file called ASSIGN.SYS. This file is read when GEM first loads into the system. Device drivers and alterations to the ASSIGN.SYS file are beyond the scope of this book. They have been presented here to clarify workstation usage. For further information, look in a book that discusses the implementation of GEM on a computer. When you open a physical workstation, you specify the physical device ID. GEM locates the device and creates a workstation for it. Only one physical workstation may be allocated to each physical device. As long as the GEM desktop is in use, it has a physical workstation open for the console. You can use this physical workstation as your program's workstation. However, if

you change any attributes, these changes remain in use when your program exits and GEM returns to the desktop. To use the system workstation, your program has to remember the setting for each attribute and restore these settings before exiting back to the desktop. This is quite cumbersome so GEM provides an easier method.

Table 3-1: Device Identification Number

Device	*Range of ID Numbers*
Monitor	1-10
Plotter	11-20
Printer	21-30
Metafile	31-40
Camera	41-50
Tablet	51-60

The virtual workstation is similar to the physical workstation in appearance. The only difference between the two is that a virtual workstation is attached to a physical workstation instead of a physical device. Any number of virtual workstations may be associated with a physical workstation. This means that your application can open a virtual workstation attached to the system workstation, change the program's workstation as required, and exit back to GEM without affecting the desktop's workstation attributes at all. Even though both workstations output to the same screen, the program's workstation defines an environment completely independent of the system workstation's environment.

Your program can open two or more virtual workstations to provide for different types of output. For example, in a computer-aided design program, you use three workstations. The first workstation might be for graphic output and use small letters, thin lines, and the color black. The second workstation might be for command and status output and use larger letters and the color blue. The third workstation might be for error messages and use the color red and perhaps a different font altogether. While your program runs, it would choose the appropriate workstation depending upon the type of output being produced. An example of multiple workstations is given in Chapter 4.

When your program has finished, it must relinquish its workstations before it returns to the desktop. This allows GEM to reclaim the memory used by the workstations and to make any physical devices available to other programs. For each workstation opened by your program, you must have a corresponding close workstation command. The system workstation remains open because it was opened by the

desktop before your program was initiated. If you were to close the system workstation, the desktop would not have an output device and the system would bomb.

The GEM Skeleton Program

As mentioned at the beginning of this chapter, use of GEM does not come without a price. Your first programming task is to enter text, compile, and link a file that performs all the overhead routines used in a GEM application. Entering and debugging this file at this time saves much time in later programming exercises and in your own applications. This outline program file is used for all the programming exercises and can be used as the basis for any GEM program you write.

Organizing the Outline

In every GEM program there is a minimum amount of overhead. In Chapter 1 the global VDI arrays are listed. At the beginning of this chapter, the basic GEM interface procedures are listed. To make programming somewhat easier, several more global variables are added. All this information in a program can produce a very confusing source code file. To avoid this, the outline program has been divided into six major sections: system header files and constants, GEM application overhead, application-specific data, GEM-related functions, application functions, and the main program.

When you are writing a GEM application, it is extremely important to keep your program organized. Because all programs in this book use this outline format, we suggest that you stick with this format for now. If you decide on a different format for your own programs, try it later. Remember that organization is the key. If you don't organize your programs, you spend more time looking through your files than you do programming.

The outline program referred to in this section, called OUTLINE.C, is shown in Listing 3-1.

Header Files

The section titled "System Header Files & Constants" lists the four files stdio.h, osbind.h, gemdefs.h, and obdefs.h. These are standard header files for the VDI used on most compilers and are used in all programs in this book. This section also defines the constant values

Listing 3-1 Program OUTLINE.C

```
/*********************************************************************
     OUTLINE.C  Outline for GEM application C program

     Use this outline to organize your C programs.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include    <stdio.h>                       /* Standard IO */
#include    <osbind.h>                      /* GEMDOS routines */
#include    <gemdefs.h>                     /* GEM AES */
#include    <obdefs.h>                      /* GEM constants */

#define     FALSE       0
#define     TRUE        !FALSE


/*************************************************
     GEM Application Overhead
*************************************************/


/* Declare global arrays for VDI. */
typedef     int     WORD;                   /* WORD is 16 bits */

WORD        contrl[12],                     /* VDI control array */
            intout[128], intin[128],        /* VDI input arrays */
            ptsin[128], ptsout[128];        /* VDI output arrays */

WORD        screen_vhandle,                 /* virtual screen workstation */
            screen_phandle,                 /* physical screen workstation */
            screen_rez,                     /* screen resolution 0,1, or 2 */
            color_screen,                   /* flag if color monitor */
            x_max,                          /* max x screen coord */
            y_max;                          /* max y screen coord */


/*************************************************
     Application Specific Data
*************************************************/


/*************************************************
     GEM-related Functions
*************************************************/
```

Listing 3-1 (continued)

```
WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}
set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);  /* mono 2, color 0 or 1 */
}


/*************************************************
     Application Functions
*************************************************/


/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                          /* application init verify */
```

Listing 3-1 (continued)

```
WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }

     screen_phandle =   /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Get screen attributes */

/*************************************************
     Application Specific Routines
*************************************************/


/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     Crawcin();                     /* GEMDOS character input */
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
```

TRUE and FALSE. These constants are defined for the sake of convenience and to make your code more readable.

The first header file included in this section, stdio.h, controls standard input and output. It includes some C definitions for functions, external declarations specifying function types, and some system constants. File stdio.h may vary from compiler to compiler. This means that you may need to include some other declarations to get your Atari program to run. If this is the case, you may add these declarations to the stdio.h file or you may put them in this section of the outline program.

The second header file, **osbind.h,** declares the function **bios( ),**

xbios( ), and **gemdos( )**, which actually interface with the operating system. Additionally, osbind.h defines the functions names for the particular BIOS, XBIOS, and GEMDOS calls using the *#define* precompiler directive. For example, the function **Settime( )** is actually an XBIOS call and is defined as this:

```
#define  Settime(a)  xbios(22, a)
```

The function **xbios( )** calls XBIOS routine number 22 with the parameter **a**. The purpose of this *#define* statement is to make your program more readable and easier to write because you don't have to remember what number to use for the set time function. A complete listing of the header files is included in Appendix B.

The last two header files, gemdefs.h and obdefs.h, primarily contain definitions for constant values used by GEM. For example, some GEM functions use one of their parameters to define the type of operation to perform. The writing modes are a good example of this. As explained earlier, the writing mode you use can be replace, transparent, XOR, or reverse transparent. As a parameter in the set writing mode, these modes have numerical values of 0, 1, 2, or 3, respectively. Instead of having to remember these values, you can use the constant names **MD_REPLACE**, **MD_TRANS**, **MD_XOR**, and **MD_ERASE**. This makes your programs much easier to read, write, and understand.

GEM Application Overhead

The second major section of the outline program contains information for GEM's application overhead. Included in this section are declarations for the variables used by the VDI and some variables you use to describe the environment the program is running in.

The first line in this section is a typedef operation. This line defines a new variable type called WORD to be equivalent to an integer. A type definition tells the computer what kind of data it is processing. All GEM documentation uses the type WORD for data processing. On the 68000 microprocessor, the length of a word is the same size as an integer, that is, 16 bits. To remain consistent with the GEM documentation, the type WORD is used throughout this book for all GEM parameters. This definition also allows for easy portability of your programs. If you move the program to another computer with a different word size, you can simply redefine WORD and recompile the program.

The first use of type WORD is to declare the global arrays used by GEM, which include **contrl[12]**, **intout[128]**, **intin[128]**, **ptsin[128]**, and **ptsout[128]**. These arrays are discussed in detail in Chapter 1.

The next set of lines under GEM Application Overhead are declarations for variables used by the programs in this book. The first of

these variables, **screen_vhandle**, is a handle to the virtual workstation for the console. The second variable, **screen_phandle**, is a handle to the physical workstation of the console. **Screen_rez** is the resolution and is either 0, 1, or 2 (low resolution, medium resolution, or high resolution, respectively). **Color_screen** is a flag that is TRUE if the program is running on a color monitor; otherwise it is FALSE. **X_max** and **y_max** are the values of the maximum coordinates in the x and y directions on the current screen.

Application-Specific Data

Because this is an outline program, it does not perform an application. Therefore, this section does not require any information. However, when you begin to write applications, this section will hold the global variables, structure definitions, and constant definitions used specifically by your application.

GEM-Related Functions

For now, this section provides a general-purpose open virtual workstation function and initialization function. Function **open_vwork( )** opens a new virtual workstation. Its input parameter is the handle to the physical workstation to which this virtual workstation is attached. Function **open_vwork( )** returns the handle of the new virtual workstation.

Here is the first use of the VDI function **v_opnvwk( )**. This function opens a virtual workstation and has three parameters: an input array of eleven elements, the address of the handle to a physical workstation, and an output array of 57 elements. The input array specifies the settings for several of the workstation attributes. The last element specifies the coordinate system to use. The output array contains information about the new workstation such as pixel size and various output capabilities. **V_opnvwk( )** returns the handle of the new virtual workstation in place of the physical workstation handle. For example, in **open_vwork( )** in the outline program, the input array **work_in[ ]** is set to specify the default values except for the coordinate system. Here the raster coordinates are used. **Open_vwork( )** calls **v_opnvwk( )** with the input array, the physical handle passed in **phys_handle**, and the output array. **Open_vwork( )** then returns the value of the new handle. If the value of the handle is 0, the **v_opnvwk( )** function is unsuccessful.

All GEM and Atari functions are listed in the appendices of this book. For this reason, whenever a new GEM or Atari function is introduced, its purpose and usage in the application program is given a detailed explanation. However, details of parameters and values

returned can be found in the appendices. In most cases, such descriptions are useless as in the case of the **v_opnvwk( )** function, because all default attributes can be set using a clearly labelled VDI function (see Chapter 4). Many of the returned values are of little use for most applications. Some of these values are used in the next function listed in program OUTLINE.C.

You may wonder why the RC system is used as a default for the screen. This is because the virtual workstation must inherit the physical workstation's coordinate system. The GEM desktop opens the screen's physical workstation using raster coordinates because the AES works with raster coordinates. Therefore, any programs you write that use the system monitor or the AES, must work in raster coordinates. If you have another monitor attached to your computer with its appropriate device driver, you can open a physical workstation for it using NDC. If you try to open a virtual workstation for the system screen with NDC, GEM automatically overrides your setting and changes it to RC.

The second function defined in the section for GEM-related functions is called **set_screen_attr( )**. This function sets the global variables **x_max**, **y_max**, **screen_rez**, and **color_screen**. To obtain the values, a call to VDI function **vq_extnd( )** is made. The **vq_extnd( )** function exemplifies the standard format for all VDI function parameter lists. All VDI function parameter lists have as their first parameter the handle to a workstation. The only functions that vary from this format are **v_opnwk( )** and **v_opnvwk( )**. As stated earlier, all VDI input and output is associated with a particular workstation. Thus, any VDI function must have a workstation handle (either physical or virtual) as its first parameter.

The **vq_extnd( )** function also has a second and third parameter. The second parameter determines the set of attribute values to be returned. If the parameter is a 0, the same values returned by the **v_opnwk( )** function are returned. If the parameter is a 1, an extended set of values is returned. The third parameter is an array of 57 elements, which holds the returned values.

In **set_screen_attr( )**, **vq_extnd( )** is called so that it returns the same values as the **v_opnwk( )** function. Then the appropriate global variables are set. These values might have been set in function **open_vwork( )** above. However, this would only allow **open_vwork( )** to be used once and only for the screen. By moving the initialization of these global variables to a separate function, you can use **open_vwork( )** to open as many virtual workstations as needed for any type of device.

The XBIOS function **Getrez( )** is used to retrieve the current screen resolution. Because of the way Atari designed the ST, low or medium resolution implies a color monitor or at least color representation in

the bit map. High resolution is available only with the monochrome monitor. Thus, if **screen_rez** is 0 or 1, **color_screen** is set to TRUE; otherwise it is set to FALSE.

Application Function

Similar to the section for application-specific data, this section holds the functions used in application programs. Because OUTLINE.C performs no action, there are no functions listed here.

The Main Program

All C programs must have a function called **main( )** where program execution begins. In the GEM programs presented throughout this book, **main( )** is used as the flow control module. It executes the list of procedures given at the beginning of this chapter. Function **main( )** is divided into three subsections: GEM access initialization, application-specific routines, and program clean-up and exit.

GEM Access Initialization

Function **main( )** begins by identifying itself to GEM through the AES function **appl_init( )**, which initializes GEM to accept the application as an active process. The AES responds by returning a positive integer value. If a negative value is returned, the AES has a problem initializing GEM. In this case, **main( )** reports the error to the user and aborts the program. The GEMDOS function **Cconws( )** writes a string to the console. A GEMDOS function is used because the program has not been properly initialized and you do not know if you have access to the VDI or AES. The GEMDOS function **Crawcin( )** waits for input from the console. It returns the keycode value pressed without echoing the character to the screen. When this function is called, the program simply pauses and waits for the user to press a key.

If GEM application initialization goes well, **main( )** continues by initializing the program's global variables. First the handle of the console's physical workstation is requested using another AES function, **graf_handle( )**. This function returns the handle of the physical workstation. It also sets its four parameters to the width and height of the system font character cell (in pixels) and the width and height of a box large enough to hold a system font character (in pixels), respectively. Because the function requires these addresses as parameters, they have been included. For most programs in this book, these values are ignored.

The last two steps of GEM access initialization are to get a virtual

workstation using **open_vwork( )** (defined earlier) and to set the remaining global values through **set_screen_attr( )** (also defined earlier). This completes the first three steps a GEM application must perform.

Application-Specific Routines

This section contains calls to the application functions defined under the section for application-specific functions. Generally these include an initialization function and several functions to process your program.

Program Clean-up and Exit

This section covers the last two steps a program must perform. First, function **Crawcin( )** is called to pause the program before it exits. The first few programs you write display some output on the screen and then exit. If your program doesn't pause before it exits, you do not see what was drawn because the first thing the desktop does is erase the screen when the program returns to the desktop. To give you time to see what the program produced, a pause is placed here before the program ends.

The next function, **v_clsvwk( )**, closes the virtual workstation opened earlier. It, too, uses the workstation handle as its first and only parameter. Note that for each workstation you open, you must have a corresponding close. You should first close all of your virtual workstations using **v_clsvwk( )**. Then close all physical workstations you opened using **v_clswk( )**. Remember not to close the desktop's physical workstation or you will have problems.

Finally, function **appl_exit( )** tells GEM that the application has terminated. This returns all program memory to the available pool and causes the desktop to be restarted.

This is basically the minimum programming you need to run a program using GEM on the Atari ST. If you want to write a quick C program using the standard input and output, you can provided you do not use any GEM VDI or AES function calls. You can use GEM BIOS or XBIOS calls freely in your C programming and you can use any standard C functions. However, the screen may act a little oddly because your program does not exactly match the GEM interface.

In reference to function origin, note that all VDI functions begin with letter "v." All GEMDOS and XBIOS functions begin with a capital letter. Standard C function and functions written specifically for this book all begin with lower-case letters. This should help you identify the origin of a function in programs presented here or elsewhere.

Now, you should enter this outline program, compile it, link it, and

run it. Nothing happens except that the program runs and the screen freezes because it is waiting for you to press something on the keyboard. When you strike a key, the desktop returns. Get the outline program working so that it compiles, links, and executes without any problems.

Kinetic Line Art

LINES.C (see Listing 3-2) is an example of programming using the outline program. This is a kinetic line art program that draws what appears to be a moving set of lines on the screen.

Listing 3-2 Program LINES

```
/*********************************************************************
     LINES.C     Draw kinetic line art

     This program demonstrates the use of the polyline functions by
     drawing lines between two pairs of moving points on the screen.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
         screen_phandle,           /* physical screen workstation */
         screen_rez,               /* screen resolution 0,1, or 2 */
         color_screen,             /* flag if color monitor */
         x_max,                    /* max x screen coord */
         y_max;                    /* max y screen coord */
```

Listing 3-2 (continued)

```
/*************************************************
     Application Specific Data
*************************************************/

/* Constant values for drawing area */
int  x_lower,                        /* lowest x value */
     y_lower,                        /* lowest y value */
     x_upper,                        /* highest x value */
     y_upper;                        /* highest y value */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)
          work_in[i] = 1;
     work_in[10] = 2;
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     v_clrwk(new_handle);                /* clear workstation */
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);     /* mono 2, color 0 or 1 */
}
```

Listing 3-2 (continued)

```
/*************************************************
     Application Functions
*************************************************/

long Rnd_rng(low, hi)
long low, hi;
/*************************************************
Function: Generate a random number between low and hi, inclusive.
Input:    low = lowest value in range.
          hi = highest value in range.
Output:   Returns random number.
*************************************************/
{
     hi++;                    /* Include hi value in range */
     return( (Random() % (hi - low)) + low);
}

inchk(x,dx,lb,ub)
int  x,dx,lb,ub;
/*************************************************
Function: This function tests whether a number (x) will be outside
     a given range if a move (dx) is added to it. If the test is
     true, the direction of movement is reversed.
Input:    x    number to test
          dx   delta to add to x
          lb   lower bound
          ub   upper bound
Output:   Returns new dx value.
*************************************************/
{
     if ( x+dx <= lb || x+dx >= ub )
          dx*=-1;
     return(dx);
}

draw_frame()
/*************************************************
Function: Draw a rounded rectangle border.
Input:    None.
Output:   None.
*************************************************/
{
WORD pts[4];                        /* array to hold corner vertices */

/* Set boundaries for border rectangle */
     pts[0] = x_lower;              /* Upper left */
     pts[1] = y_lower;
     pts[2] = x_upper;              /* Lower right */
     pts[3] = y_upper;
     v_rbox(screen_vhandle, pts);   /* Draw a rounded box */
     return;
}
```

Listing 3-2 (continued)

```
draw_lines()
/*************************************************
Function: Do kinetic line art
Input:    None.
Output:   None.
*************************************************/
{
int  line_number,                    /* loop control */
     k;

WORD px1, py1,                       /* endpoint 1 for line p */
     px2, py2,                       /* endpoint 2 for line p */
     pdx1, pdx2,                     /* delta x for endpoints */
     pdy1, pdy2;                     /* delta y for endpoints */

WORD rx1, ry1,                       /* variables for line r */
     rx2, ry2,
     rdx1, rdx2,
     rdy1, rdy2;

WORD pxy[100], rxy[100];             /* arrays to hold 2 sets of 25 lines */

do                                   /* begin screen control loop */
{
     v_clrwk(screen_vhandle);        /* clear screen */
     draw_frame();                   /* draw border */
/* Initialize line endpoints */
/* Line P endpoints */
     px1 = Rnd_rng( (long)x_lower, (long)x_upper);
     py1 = Rnd_rng( (long)y_lower, (long)y_upper);
     px2 = Rnd_rng( (long)x_lower, (long)x_upper);
     py2 = Rnd_rng( (long)y_lower, (long)y_upper);
     pdx1 = Rnd_rng( -10L, 10L);
     pdy1 = Rnd_rng( -10L, 10L);
     pdx2 = Rnd_rng( -10L, 10L);
     pdy2 = Rnd_rng( -10L, 10L);
     rx1 = Rnd_rng( (long)x_lower, (long)x_upper);
     ry1 = Rnd_rng( (long)y_lower, (long)y_upper);
     rx2 = Rnd_rng( (long)x_lower, (long)x_upper);
     ry2 = Rnd_rng( (long)y_lower, (long)y_upper);
     rdx1 = Rnd_rng( -10L, 10L);
     rdy1 = Rnd_rng( -10L, 10L);
     rdx2 = Rnd_rng( -10L, 10L);
     rdy2 = Rnd_rng( -10L, 10L);

/* Each point requires 4 elements in the pxy or rxy array. The current
     line drawn is held in elements 96, 97, 98, and 99 for the x and y
     coordinates of the first endpoint and the x and y coordinates of
     the second endpoint, respectively. The next line to be erased is
     in elements 0, 1, 2, and 3.
*/
```

Listing 3-2 (continued)

```
     for( k=0; k<100; k++ )
          pxy[k] = rxy[k] = 0;      /* clear arrays */
     vsl_color(screen_vhandle,1);  /* Set color to black */
/* Line drawing loop begins here */
/* Change loop end value to draw any number of lines */
     do
     {
          rxy[96] = rx1; rxy[97] = ry1;  /* Set next line to */
          rxy[98] = rx2; rxy[99] = ry2;  /* be drawn */
          pxy[96] = px1; pxy[97] = py1;
          pxy[98] = px2; pxy[99] = py2;
          v_pline(screen_vhandle, 2, &pxy+96);    /* Draw polyline using */
          v_pline(screen_vhandle, 2, &rxy+96);    /* 2 vertices (one line) */

          v_pline(screen_vhandle, 2, &pxy);  /* Redraw first line */
          v_pline(screen_vhandle, 2, &rxy);  /* to erase it */

          for( k=0; k<96; k++)          /* Shift endpoints in arrays */
          {
          pxy[k] = pxy[k+4];
          rxy[k] = rxy[k+4];
          }

          /* Calculate endpoints of next lines */
          px1 += pdx1;   py1 += pdy1;
          px2 += pdx2;   py2 += pdy2;
          rx1 += rdx1;   ry1 += rdy1;
          rx2 += rdx2;   ry2 += rdy2;
          rdx1 = inchk(rx1, rdx1, x_lower, x_upper);
          rdx2 = inchk(rx2, rdx2, x_lower, x_upper);
          rdy1 = inchk(ry1, rdy1, y_lower, y_upper);
          rdy2 = inchk(ry2, rdy2, y_lower, y_upper);
          pdx1 = inchk(px1, pdx1, x_lower, x_upper);
          pdx2 = inchk(px2, pdx2, x_lower, x_upper);
          pdy1 = inchk(py1, pdy1, y_lower, y_upper);
          pdy2 = inchk(py2, pdy2, y_lower, y_upper);
     } while (!Cconis());                /* check if key pressed */
} while ((Crawcin() & 0x7F) != 27);           /* escape key exits */
     return;
}

/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;
```

Listing 3-2 (continued)

```
/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();          /* Initialize AES routines */
     if (ap_id < 0)                /* no calls can be made to AES */
     {                             /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                /* set exit value to show error */
     }
     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */

/*************************************************
     Application Specific Routines
*************************************************/

     vsl_type(screen_vhandle, 1);  /* Set polyline pattern to solid */
     vsl_width(screen_vhandle, 1); /* Set line width */
     vsf_color(screen_vhandle, 0); /* Set fill color to 0 (white) */
     vsf_perimeter(screen_vhandle, TRUE);    /* Turn on perimeter */
     vswr_mode(screen_vhandle, MD_XOR);
/* set boundaries */
     x_lower = 10;
     y_lower = 10;
     x_upper = x_max - 10;
     y_upper = y_max - 10;
/* start program */
     draw_lines();                 /* Do line art */

/*************************************************
     Program Clean-up and Exit
*************************************************/

     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
```

Each line consists of two endpoints. Each endpoint has an x and a y coordinate. To make kinetic line art, the program draws a line between the two endpoints and then moves the two endpoints. Another line is drawn and the endpoints are moved again. Then another line is drawn and another moved. After a specified number of lines are drawn (in the case of this program, 25), the first line is erased before another line is drawn. Then another line is erased and

another line drawn. This process continues until the user presses the Escape key.

The important aspects to look for in this program are its organization and the use of the writing mode XOR. The program builds on the OUTLINE.C file following its intended flow quite carefully. The use of the XOR writing mode is used in many programs throughout this book.

As an example of the efficiency of using the XOR writing mode, assume that you have a blank monochrome screen where all bits are 0. Now, by drawing a line, you are writing 1 bits onto 0 bits. Under the XOR operator, this results in a 1 bit at each location under the line in the bit map. Therefore, a line is displayed on the screen. Now draw another line directly on top of the first line. Each 1 bit on the line is XORed with the 1 bit in the bit map. At each location in the bit map, a 0 results. The end result is that you have erased the first line. By drawing a line on a blank screen, you get a line to appear. By drawing the same line in the same place, you get the line to disappear. This is the unique feature of the XOR operator: if you apply the same XOR operation twice in a row, you wind up where you started.

This fact about the XOR writing mode is used in program LINES. The program simply stores the endpoints for the 25 lines shown on the screen. Before the next line is drawn, the first line is redrawn to erase it. Then the next line is drawn and erased before a new line is drawn. This process can go on and on. The application-specific data of Listing 3-2 holds four variables: **x_lower**, **y_lower**, **x_upper**, and **y_upper**. The variables hold the lowest and highest x and y coordinate values that can be used. These values set the range over which the lines can travel.

Function **main( )** gives an overview of the program's flow. As expected, the first thing the program does is initialize GEM access. After this is done, the application begins its processing.

Under the application-specific routines, the first five lines set the workstation attributes to be used by the program. The first function, **vsl_type( )**, sets the line pattern to be a continuous or solid line. Function **vsl_width( )** sets the line width to 1 unit (in this case a pixel). The **vsf_color( )** function sets the fill color to the background color (white). The fill color is the color used when a polygon is filled. The next function, **vsf_perimeter( )**, tells the VDI to draw a perimeter when it draws a predefined shape. The last attribute function, **vswr_mode( )**, sets the writing mode to XOR. Note the use of the defined constant MD_XOR from the osbind.h file. Do not worry about fully understanding how these attribute functions work. The program in Chapter 4 fully uses the workstation output and attributes.

The next step after getting the workstation attributes is setting the

boundaries of the line-drawing routine. The program arbitrarily defines the boundaries to fall within 10 pixels from each edge. Since the lowest coordinate value is already known to be 0, the program sets **x_lower** and **y_lower** to the value 10. Because the upper coordinate values are not known until the program is running, the program uses the global values from **x_max** and **y_max** to determine the upper limits. These values depend on the type of monitor and resolution used. Once the boundary values have been set, the program begins by calling function **draw_lines( )**.

Function **draw_lines( )** returns when the Escape key has been pressed. Because the user has requested the program exit, it is safe to assume that the images have been drawn on the screen. Therefore, the **Crawcin( )** pause has been removed from this program and normal program exiting is performed.

Function **draw_lines( )** controls the program flow while lines are being drawn on the screen. The program listed here actually keeps track of two sets of lines moving independently on the screen. The flow of **draw_lines( )** follows this outline:

```
do begin the line drawing function
   clear the screen
   draw a border
   randomly select the endpoints for lines P and R
   randomly select the movement for each endpoint
   clear the storage arrays for each set of lines
   set the line drawing color to black
   do begin the line drawing output
      save the next lines to be drawn in the storage arrays
      draw lines P and R
      redraw the first lines in the storage arrays for P and R
      move each endpoint
   until a key is pressed
until the key pressed is the Escape key
```

Starting at the top of this outline, the entire **draw_lines( )** function is one do-while loop, which iterates each time a key is pressed. If the key is not the Escape key, the loop repeats. Thus, by pressing any key other than the Escape key, the user can clear the screen and start the line-drawing sequence over again.

The remainder of **draw_lines( )** consists of line-drawing initialization and a line-drawing loop. The initialization process begins by clearing the screen using the **v_clrwk( )** function. A border is drawn by the application function **draw_frame( )**. Looking at **draw_frame( )**, you see an array being initialized and the VDI function **v_rbox( )**.

When drawing a rectangle, you only need to specify the coordinates of the upper left corner and lower right corner. This is done in array **pts[ ]**. As discussed in Chapter 1, **pts[ ]** follows the point array format where the first x coordinate is placed in element 0 followed by its corresponding y coordinate in element 1. The next point's x and y coordinates are stored in elements 2 and 3, respectively. The function **v_rbox( )** draws a filled rectangle with rounded corners.

Back in **draw_lines( )** after the call to **draw_frame( )**, the next step is to initialize the endpoints of the two lines and each endpoint's movement. The initial values are chosen through the application function **Rnd_rng( )**. Function **Rnd_rng( )** is the first function defined in the application function section of program LINES. This function returns a *long* integer value within the range specified by parameter low and hi. The range includes the values low and hi. **Rnd_rng( )** uses the XBIOS function **Random( )**. Function **Random( )** returns a 24-bit random value. Function **Rnd_rng( )** takes this value and performs a modulo operation to get a value between 0 and the difference between **hi** and **low**. By adding the value **low** to the modulo result, **Rnd_rng( )** produces a random number within the range requested. The XBIOS function **Random( )** is different from the C function **rand( )** in that you cannot set the seed for **Random( )** to specify a sequence.

Once **draw_lines( )** has set the initial values for the endpoints and their movement, it clears the arrays that keep track of the lines on the screen. Each line requires two endpoints and each endpoint uses two values. Thus, one line is defined by four elements in the array. To store 25 different lines, 100 elements are required for each array. After clearing the arrays, the line color is set and the line-drawing loop begins.

The VDI function to draw a line, **v_pline( )**, is actually a polyline-drawing routine. Given a set of points, **v_pline( )** draws a line from the first point to the second point to the third point and so on until the number of specified points has been reached. The points are passed to **v_pline( )** in an array using the standard VDI point format. For this program, only two points are specified because only one line is being drawn at a time.

The next line to be drawn is stored in elements 96, 97, 98, and 99 of the line storage arrays. Using the fact that the array name is the base address of the array, the **v_pline( )** function calls in **draw_line( )** pass the address of the 96th element in the array. Thus, the newest lines P and R are drawn.

The oldest line showing on the screen is stored in elements 0, 1, 2, and 3. By passing the base address of the array to **v_pline( )**, the program redraws the oldest lines on the screen. By redrawing these lines in the XOR writing mode, the program effectively erases the

oldest lines. Note that until 25 lines are drawn, the values in elements 0, 1, 2, and 3 are all 0. Therefore, when **v_pline( )** tries to draw a line from (0,0) to (0,0) it simply draws a point (a line with no length) at coordinate (0,0).

The next step for **draw_line( )** is to shift the position of the endpoints in the arrays. Because each line takes four points, each point must be shifted down by four elements. This is done in the for-loop shown in the listing.

The final step for **draw_line( )** is to calculate the endpoints for the next pair of lines. This is done by adding the appropriate movement value to each coordinate value. After the movement values are added, the coordinate values are checked using application function **inchk( )**. Function **inchk( )** tests if the next addition of the movement value causes the coordinate to move beyond the specified range. Function **inchk( )** is passed the current coordinate, the current movement value, the lowest range value, and the highest range value. If the next coordinate value remains within the range, the **inchk( )** returns the current movement value. If the next coordinate value exceeds one of the range limits, **inchk( )** returns the negative movement value. This makes it appear that the endpoints are "bouncing off" the walls of the border. For example, suppose the x coordinate limits are 10 and 630 (as on a monochrome monitor). If **px1** has the value of 150 and **pdx1** has the value of −4, the next value of **px1** is 144, which is within the range, so **inchk( )** returns the value of −4. If **px1** has the value of 12, the next value of **px1** is 8, which is too low. In this case, **inchk( )** returns 4, so that the next time **px1** is changed, it is changed by 4 instead of −4.

The end of the inside do-while loop uses the GEMDOS function **Cconis( )** to test if the console has a character waiting to be read (that is, the user has pressed a key). If the function returns FALSE, no key has been pressed and another line is drawn. If the function returns TRUE, the user has pressed a key and the inner loop exits.

The outer do-while loop checks which key has been pressed by using the value returned from function **Crawcin( )**. Function **Crawcin( )** reads the current character waiting to be read from the console and returns the corresponding keycode. The keycode on the Atari ST consists of a 16-bit value. For ASCII characters (the first 128 characters), the lower 7 bits contain the ASCII value. By performing a bit-wise AND operation on the value returned from **Crawcin( )**, the program gets the ASCII value of the key pressed. Since the ASCII value of Escape is 27, the loop begins again if the value returned is not 27. If Escape is pressed, the function **draw_lines( )** exits and returns to **main( )**. A listing of the full keycodes is given in Appendix C.

This completes the description of program LINES. Enter and com-

pile this program to practice using the OUTLINE program and to get a feel for programming in GEM. When you have LINES running, make some changes to it. First try changing some of the range values for the endpoint initialization. Then try using the replace writing mode instead of the XOR writing mode. To do this, draw the new line using line color 1 (black) and the old line using line color 0 (background). This means that you need to use **vsl_color( )** to set the line-drawing color before each **v_pline( )** call.

CHAPTER FOUR

VDI Output and Friends

All VDI functions produce output on a workstation. A workstation has a logical bit map on which all output is produced. Whether or not a physical bit map is used is irrelevant, since the output is always the same no matter which device is used to produce the picture. That's the principle behind using VDI.

The general procedure used in a GEM program is to open a workstation (either physical or virtual) and output the graphic drawings to it. All VDI functions (including input, output, and workstation attributes) use a workstation handle as their first parameter.

The Workstation Workout

How do you go about using all these functions? Program GRAFDEMO exercises the various functions used to produce output and set the workstation attributes. Go directly to function **main( )** in Listing 4-1 to see the general flow of the GRAFDEMO program. The preliminary setup is the same as in the two previous programs, OUTLINE and LINES. First a virtual workstation is opened, and then the global system variables are set.

There are four primary application functions used to demonstrate the various workstation attributes: **draw_line( )**, **draw_rect( )**, **draw_circ( )**, and **draw_text( )**. Each works with a different type of output. Function **draw_line( )** demonstrates various line-drawing capabilities and routines and attributes related to drawing lines. The function **draw_rect( )** demonstrates drawing rectangles and their as-

Listing 4-1 Program GRAFDEMO

```
/*********************************************************************
     GRAFDEMO.C     Demonstrate VDI graphics routines.

     VDI graphics routines are explained in the text. Change this
     program to test different combinations of graphic attribute
     settings.
*********************************************************************/

/*************************************************
    System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int WORD;                /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

/*************************************************
     Application-Specific Data
*************************************************/

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
```

Listing 4-1 (continued)

```
{
WORD work_in[11],
        work_out[57],
        new_handle;                     /* handle of workstation */
int     i;

     for (i = 0; i < 10; i++)           /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                   /* use raster coords */
     new_handle = phys_handle;          /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}


/*************************************************
     Application Functions
*************************************************/

calc_shape(num_pts, a)
WORD *num_pts, a[];
/*************************************************
Function: Used by draw_line() to calculate an arrow given
     the leftmost vertex.
Input:    a[0] = x-coord of point.
          a[1] = y-coord of point.
Output:   Returns array a filled with points for v_pline
     function to draw an arrow with six line segments.
     Num_pts contains the number of points in the shape.
*************************************************/
{
/* SHAPE_SIZE determines the size of the arrow */
#define   SHAPE_SIZE     18

     *num_pts = 7;
```

Listing 4-1 (continued)

```
/* The arrow is draw from the upper leftmost point. */
     a[2] = a[0] + SHAPE_SIZE;
     a[3] = a[1];
     a[4] = a[2];
     a[5] = a[3] - SHAPE_SIZE;
     a[6] = a[2] + SHAPE_SIZE;
     a[7] = a[1] + (SHAPE_SIZE/2);
     a[12] = a[0];
     a[13] = a[1] + SHAPE_SIZE;
     a[10] = a[2];
     a[11] = a[13];
     a[8] = a[4];
     a[9] = a[11] + SHAPE_SIZE;
     return;
}


draw_line()
/*************************************************
Function: Demonstrate VDI line drawing functions.
Input:    None.
Output:   None.
*************************************************/
{
WORD count, pxy[32];
WORD x[16], y[16];                  /* start points for shapes */
int  i;

     x[1] = 10;     y[1] = 20;     /* set start points 1st  ine */
     x[2] = 70;     y[2] = 20;
     x[3] = 130;    y[3] = 20;
     x[4] = 190;    y[4] = 20;

     x[5] = 10;     y[5] = 85;     /* set start points 2nd line */
     x[6] = 70;     y[6] = 85;
     x[7] = 130;    y[7] = 85;
     x[8] = 190;    y[8] = 85;

     x[9] = 10;     y[9] = 150;    /* set start points 3rd line */
     x[10] = 70;    y[10] = 150;
     x[11] = 130;   y[11] = 150;
     x[12] = 190;   y[12] = 150;


/* Show change in line width */
     v_clrwk(screen_vhandle);       /* clear screen */
     for (i = 1; i <= 12; i++)
     {
          pxy[0] = x[i];            /* set start points */
          pxy[1] = y[i];
          calc_shape(&count, pxy); /* set arrow */
```

Listing 4-1 (continued)

```
    vsl_width(screen_vhandle, 1);           /* set line width */
    v_pline(screen_vhandle, count, pxy);    /* draw line */
}
Crawcin();

/* Show change in line type */
    v_clrwk(screen_vhandle);       /* clear screen */
    vsl_width(screen_vhandle, 1); /* set default width */
    for (i = 1; i <= 12; i++)
    {
        pxy[0] = x[i];             /* set start points */
        pxy[1] = y[i];
        calc_shape(&count, pxy); /* set arrow */
        vsl_type(screen_vhandle, i);            /* set line type */
        v_pline(screen_vhandle, count, pxy);    /* draw line */
    }
    Crawcin();

/* Show change in line end style */
    v_clrwk(screen_vhandle);       /* clear screen */
    vsl_width(screen_vhandle, 9); /* set medium width */

    pxy[0] = 50;  pxy[1] = 20;     /* draw squared ends */
    pxy[2] = 150; pxy[3] = 20;
    vsl_ends(screen_vhandle, 0, 0);
    v_pline(screen_vhandle, 2, pxy);

    pxy[0] = 50;  pxy[1] = 60;     /* draw arrow ends */
    pxy[2] = 150; pxy[3] = 60;
    vsl_ends(screen_vhandle, 1, 1);
    v_pline(screen_vhandle, 2, pxy);

    pxy[0] = 50;  pxy[1] = 100;    /* draw rounded ends */
    pxy[2] = 150; pxy[3] = 100;
    vsl_ends(screen_vhandle, 2, 2);
    v_pline(screen_vhandle, 2, pxy);
    Crawcin();

/* Show change in marker type */
    v_clrwk(screen_vhandle);       /* clear screen */
    for (i = 1; i <= 12; i++)
    {
        pxy[0] = x[i];
        pxy[1] = y[i];
        calc_shape(&count, pxy);
        vsm_type(screen_vhandle, i);
        v_pmarker(screen_vhandle, count, pxy);
    }
    Crawcin();

    return;
}
```

Listing 4-1 (continued)

```
draw_boxes()
/*************************************************
Function: Draws rectangles used VDI routines.
Input:    None.
Output:   None.
*************************************************/
{
WORD pxy[4];

     v_clrwk(screen_vhandle);
     vsf_perimeter(screen_vhandle, FALSE);
     pxy[1] = 30;   pxy[3] = 90;
     pxy[0] = 30;   pxy[2] = 60;
     vr_recfl(screen_vhandle, pxy);
     pxy[0] += 50;  pxy[2] += 50;
     v_rbox(screen_vhandle, pxy);
     pxy[0] += 50;  pxy[2] += 50;
     v_rfbox(screen_vhandle, pxy);
     pxy[0] += 50;  pxy[2] += 50;
     v_bar(screen_vhandle, pxy);

     vsf_perimeter(screen_vhandle, TRUE);
     pxy[1] = 130;  pxy[3] = 190;
     pxy[0] = 30;   pxy[2] = 60;
     vr_recfl(screen_vhandle, pxy);
     pxy[0] +=  50; pxy[2] += 50;
     v_rbox(screen_vhandle, pxy);
     pxy[0] += 50;  pxy[2] += 50;
     v_rfbox(screen_vhandle, pxy);
     pxy[0] += 50;  pxy[2] += 50;
     v_bar(screen_vhandle, pxy);

     return;
}


draw_rect()
/*************************************************
Function: Demonstrate VDI rectangle & area functions.
Input:    None.
Output:   None.
*************************************************/
{
WORD pxy[32];
int  i;


/* This first draw_boxes() call uses the attribute values
*    previously set by draw_line().
*/
     draw_boxes();
     Crawcin();
```

Listing 4-1 (continued)

```
/* Reset to default values */
     vsl_width(screen_vhandle, 1);        /* set default width */
     vsl_ends(screen_vhandle, 0, 0);      /* set squared ends */
     vsl_type(screen_vhandle, 1);         /* set solid lines */
     draw_boxes();
     Crawcin();

/* Fill attribute settings */
     vsf_interior(screen_vhandle, 0);     /* hollow (default) */
     draw_boxes();
     Crawcin();

     vsf_interior(screen_vhandle, 1);     /* solid */
     draw_boxes();
     Crawcin();

     vsf_interior(screen_vhandle, 2);     /* use patterns */
     draw_boxes();
     Crawcin();

     vsf_interior(screen_vhandle, 3);     /* use hatches */
     draw_boxes();
     Crawcin();

/* Display patterns */
     v_clrwk(screen_vhandle);
     vsf_interior(screen_vhandle, 2);
     for (i = 0; i < 32; i++)
     {
          vsf_style(screen_vhandle, i+1);
          pxy[0] = ( (i%8) * 30) + 20;
          pxy[1] = ( (i/8) * 30) + 20;
          pxy[2] = pxy[0] + 20;
          pxy[3] = pxy[1] + 20;
          vr_recfl(screen_vhandle, pxy);
     }
     Crawcin();

/* Display hatches */
     v_clrwk(screen_vhandle);
     vsf_interior(screen_vhandle, 3);
     for (i = 0; i < 32; i++)
     {
          vsf_style(screen_vhandle, i+1);
          pxy[0] = ( (i%8) * 30) + 20;
          pxy[1] = ( (i/8) * 30) + 20;
          pxy[2] = pxy[0] + 20;
          pxy[3] = pxy[1] + 20;
          vr_recfl(screen_vhandle, pxy);
     }
     Crawcin();
```

Listing 4-1 (continued)

```
/* Fill area fills complex polygons. The shape in pxy array
*    is a bowtie.
*/
     v_clrwk(screen_vhandle);
     vsf_perimeter(screen_vhandle, TRUE);     /* turn on perimeter */
     vsf_interior(screen_vhandle, 2);         /* use pattern fill */
     vsf_style(screen_vhandle, 9);            /* brick pattern */
     pxy[0] = 30;  pxy[1] = 30;
     pxy[2] = 150; pxy[3] = 150;
     pxy[4] = 150; pxy[5] = 30;
     pxy[6] = 30;  pxy[7] = 150;
     pxy[8] = 30;  pxy[9] = 30;
     v_fillarea(screen_vhandle, 5, pxy);
     Crawcin();

/* Contour fill fills an area already shown on the display */
     v_clrwk(screen_vhandle);
     vsf_interior(screen_vhandle, 0);         /* use hollow fill */
     vswr_mode(screen_vhandle, MD_TRANS);     /* transparent mode */
     pxy[0] = 30;  pxy[1] = 30;               /* draw overlapping */
     pxy[2] = 100; pxy[3] = 100;              /* rectangles */
     v_bar(screen_vhandle, pxy);
     pxy[0] = 80;  pxy[1] = 80;
     pxy[2] = 150; pxy[3] = 150;
     v_bar(screen_vhandle, pxy);
     vsf_interior(screen_vhandle, 3);         /* use hatch fill */
     vsf_style(screen_vhandle, 3);            /* grid hatch */
     v_contourfill(screen_vhandle, 90, 90, -1);
     Crawcin();

}

draw_circ()
/*************************************************
Function: Demonstrate VDI circle, ellipse, & arc functions.
Input:    None.
Output:   None.
*************************************************/
{
/* draw circle and ellipse */
     v_clrwk(screen_vhandle);
     vsf_interior(screen_vhandle, 2);    /* pattern fill */
     vsf_style(screen_vhandle, 17);      /* use wavy pattern */
     v_circle(screen_vhandle, 50, 100, 40);
     v_ellipse(screen_vhandle, 150, 100, 40, 75);
     Crawcin();

/* draw circle arc and ellipse arc */
     v_clrwk(screen_vhandle);
     v_arc(screen_vhandle, 50, 100, 40, 800, 1800);
     v_ellarc(screen_vhandle, 150, 100, 40, 75, 2000, 3500);
     Crawcin();
```

Listing 4-1 (continued)

```
/* draw circle pie and ellipse pie */
     v_clrwk(screen_vhandle);
     v_pieslice(screen_vhandle, 50, 100, 40, 1800, 2100);
     v_ellpie(screen_vhandle, 150, 100, 40, 75, 3500, 500);
     Crawcin();

}

draw_text()
/*************************************************
Function: Demonstrate VDI text drawing functions.
Input:    None.
Output:   None.
*************************************************/
{
int i;
WORD char_height, char_width, cell_height, cell_width;
WORD hor_out, vert_out;
WORD attrib[10];
char s[32];

/* show plain text output, justified output, and rotation */
     v_clrwk(screen_vhandle);
     v_gtext(screen_vhandle, 30, 40, "This is v_gtext.");
     v_justified(screen_vhandle, 30, 70, "This is v_justified",
          200, FALSE, FALSE);      /* no spacing changes */
     v_justified(screen_vhandle, 30, 100, "This is v_justified",
          200, FALSE, TRUE);       /* intercharacter spacing */
     v_justified(screen_vhandle, 30, 130, "This is v_justified",
          200, TRUE, FALSE);       /* interword spacing */
     v_justified(screen_vhandle, 30, 160, "This is v_justified",
          200, TRUE, TRUE);        /* both adjustments */
     vst_rotation(screen_vhandle, 900);
     v_gtext(screen_vhandle, 300, 180, "Text on edge.");
     vst_rotation(screen_vhandle, 1800);
     v_gtext(screen_vhandle, 280, 180, "Upsidedown text.");
     vst_rotation(screen_vhandle, 0);
     Crawcin();

/* show current settings */
     v_clrwk(screen_vhandle);
     vqt_attributes(screen_vhandle, attrib);
     sprintf(s, "Current text face: %d", attrib[0]);
     v_gtext(screen_vhandle, 10, 20, s);
     sprintf(s, "Current height   : %d", attrib[7]);
     v_gtext(screen_vhandle, 10, 50, s);
     Crawcin();

/* show character height in points and absolute mode */
/* 1 point & 1 pixel */
     v_clrwk(screen_vhandle);
```

Listing 4-1 (continued)

```
     vst_point(screen_vhandle, 1, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 100, 'This is 1 point.');

     vst_height(screen_vhandle, 1, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 190, 'This is 1 pixel.');
     Crawcin();

/* Default value in point and pixel */
     v_clrwk(screen_vhandle);
     vst_point(screen_vhandle, attrib[7], &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 100, 'This is default point.');

     vst_height(screen_vhandle, attrib[7], &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 190, 'This is default pixel.');
     Crawcin();

/* 10 point & 10 pixel */
     v_clrwk(screen_vhandle);
     vst_point(screen_vhandle, 10, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 100, 'This is 10 point.');
     vst_height(screen_vhandle, 10, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 190, 'This is 10 pixels.');
     Crawcin();

/* 40 point & 40 pixel */
     v_clrwk(screen_vhandle);
     vst_point(screen_vhandle, 40, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 100, 'This is 40 point.');
     vst_height(screen_vhandle, 40, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 190, 'This is 40 pixels.');
     Crawcin();

/* 72 point & 72 pixel */
     v_clrwk(screen_vhandle);
     vst_point(screen_vhandle, 72, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 100, 'This is 72 point.');

     vst_height(screen_vhandle, 72, &char_width, &char_height,
          &cell_width, &cell_height);
     v_gtext(screen_vhandle, 10, 190, 'This is 72 pixels.');
     vst_height(screen_vhandle, attrib[7], &char_width, &char_height,
          &cell_width, &cell_height);
     Crawcin();
```

Listing 4-1 (continued)

```
/* Font variations */
     v_clrwk(screen_vhandle);
     for (i = 0; i < 20; i++)
     {
          vst_font(screen_vhandle, i);          /* set font */
          vqt_name(screen_vhandle, i, s);       /* get font name */
          v_gtext(screen_vhandle, (i/10)*150+20, (i%10)*19+20, s);
     }
     vst_font(screen_vhandle, attrib[0]);       /* return to system font */
     Crawcin();


/* Text alignment */
     v_clrwk(screen_vhandle);
/* standard */
     v_gtext(screen_vhandle, 30, 20, "Ny__");
/* vertical half line */
     vst_alignment(screen_vhandle, 0, 1, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 80, 20, "Hy__");
/* vetical ascent line */
     vst_alignment(screen_vhandle, 0, 2, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 130, 20, "Ay__");
/* vertical bottom line */
     vst_alignment(screen_vhandle, 0, 3, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 180, 20, "By__");
/* vertical descent line */
     vst_alignment(screen_vhandle, 0, 4, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 230, 20, "Dy__");
/* vertical top line */
     vst_alignment(screen_vhandle, 0, 5, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 280, 20, "Ty__");
/* horizontal left */
     vst_alignment(screen_vhandle, 0, 0, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 100, 70, "Hleft_y");
/* horizontal center */
     vst_alignment(screen_vhandle, 1, 0, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 100, 100, "Hcenter_y");
/* horizontal right */
     vst_alignment(screen_vhandle, 2, 0, &hor_out, &vert_out);
     v_gtext(screen_vhandle, 100, 130, "Hright_y");
     vst_alignment(screen_vhandle, 0, 0, &hor_out, &vert_out);
     Crawcin();


/* Show text effects */
     v_clrwk(screen_vhandle);
     for (i = 0; i < 64; i++)
     {
          vst_effects(screen_vhandle, i);
          v_gtext(screen_vhandle, (i/8)*35+20, (i%8)*22+20, "Aby");
     }
     vst_effects(screen_vhandle,0);      /* reset to normal */
     Crawcin();
}
```

Listing 4-1 (continued)

```
/*************************************************
     Main Program
*************************************************/

main()
{
int ap_id;                          /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws('***> Initialization Error. <***\n');
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }
     screen_phandle =               /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Set global screen values */

/*************************************************
     Application Specific Routines
*************************************************/

     draw_line();                   /* demonstrate line types */
     draw_rect();                   /* rectangle, bar */
     draw_circ();                   /* circle, ellipse, arc, pie */
     draw_text();                   /* text functions */

/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
/*************************************************/
```

sociated attributes. The **draw_circ( )** function shows how to draw circles, ellipses, arcs, and pies. Finally, **draw_text( )** shows the various text-drawing capabilities provided by the VDI.

Line 'Em Up: Function draw_line( )

The **draw_line( )** function is listed in the application functions section of GRAFDEMO. Basically, **draw_line( )** uses **v_pline( )** to draw a series of lines that form an arrow (see Figure 4-1). To place the arrow conveniently at any location on the screen, **draw_line( )** calls function **calc_shape( )** to fill in the array of points that define the arrow. Function **calc_shape( )** has two parameters. The first is an address to a variable that holds the total number of points. The second is an array into which the points are placed. The first two elements of this array are set to the coordinates of the first point of the shape. The remaining points are calculated with respect to the location of the initial point. By passing new starting coordinates to **calc_shape( )**, the calling function (in this case, **draw_line( )**) can place the shape at any location on the screen. The reason for having a parameter to hold the total number of points is so that **calc_shape( )** can be changed to produce any shape. By changing the **calc_shape( )** function, the **draw_line( )** function can draw any shape you desire. Of course the points array must be large enough to hold the shape. The declared constant SHAPE_SIZE determines the overall size of the shape created in **calc_shape( )**.

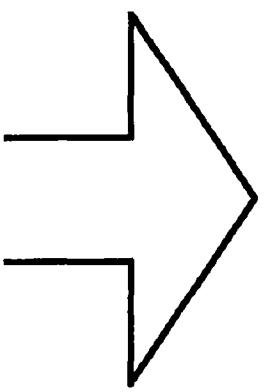


Figure 4-1 Pointed Arrow Shape

In **draw_line( )**, there are two arrays labelled x and y. These arrays hold the starting x and y coordinates of the 12 arrows to be drawn. To draw the arrows, the workstation is cleared using **v_clrwk( )**. Then for each of the 12 starting points, **draw_line( )** sets the first two elements in array pxy, calls **calc_shape( )**, and passes it the addresses

of the variable **count** and the **pxy** array. Function **draw_line( )** can now output the arrow using the current workstation attributes.

In the section that shows the change in line width, the function **vsl_width( )** is used to set the line width to values 1 through 12. The width of the line is set to **i** pixels wide. When the program is executed, only odd values are accepted (see Figure 4-2). Even values of **i** are set to the next lowest width available. When all 12 arrows have been drawn to show the line-width setting, the program pauses to wait for a key to be pressed at function **Crawcin( )**.

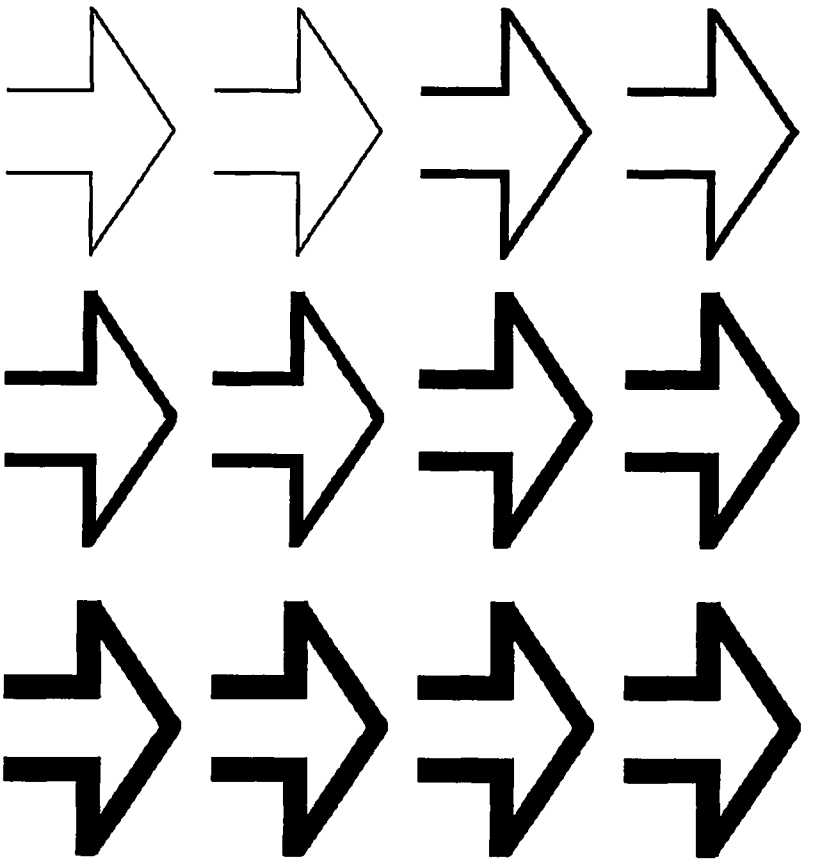


Figure 4-2 Arrows Showing Different Line Widths

The next set of variations shows the different line styles (see Figure 4-3). First, the workstation is cleared and the line width is reset to the default width (one pixel wide). The loop again produces 12 **vsl_type( )** arrow calls to set the line type to each of the 12 values. The line types are indexed from 0 to a *device-dependent value.* This means that not every output device can produce the same number of different line

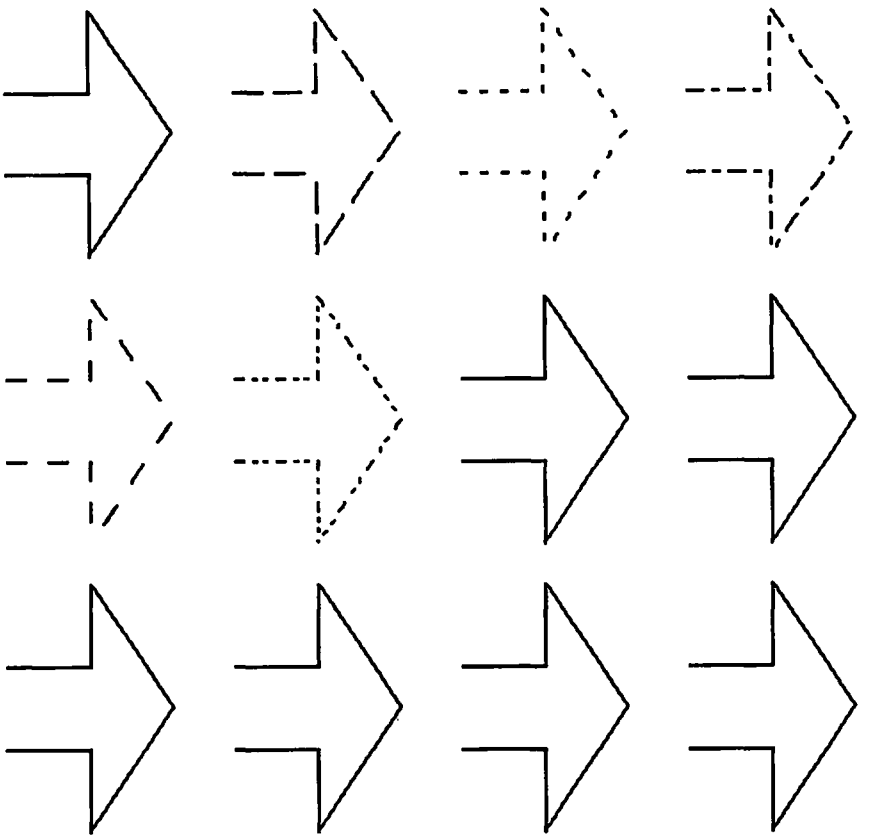


Figure 4-3 Arrows Showing Different Line Styles

styles. However, all devices have at least six different line styles to choose from (see Figure 4-4). Line type one on the screen, for instance, is a solid line. Type two is a long dash; type three is a dotted line; type four is a dash-dot line; type five is a dash; and line type six is a dash-dot-dot. There is also a seventh line style that may be defined by the user. Some devices may have the ability to support even more line styles. Any line styles above line index seven are strictly device-dependent.

The line width is set back to one because a particular output device may not be able to produce line styles with a width greater than one. In particular, if you try to put a line style on the screen with a width greater than one, *only* solid lines are produced. The screen driver does not have the capability to produce thickened line styles. If you want to draw a specific nonsolid line, it must have a width of one.

After another pause at **Crawcin( ), draw_line( )** continues to show the various line-end styles. The screen is cleared and the line width

	Style		16 bits Bit 15 Bit 0
1	_____	solid	1111111111111111
2	_ _ _ _	long dash	1111111111110000
3	. . . . .	dot	1110000011100000
4	_ . _ . _ .	dash, dot	1111111000111000
5	_ _ _ _	dash	1111111100000000
6	. . . . .	dash, dot, dot	1111000110011000
7	_ _ _ _ _	user-defined	1111000000001111

8 - n Device dependent line styles

Figure 4-4 Line Styles for Atari ST

is set to 9. Three lines are then drawn to show the three line-end styles: squared, arrowed, and rounded. Function **vsl_ends( )** sets the line-end style. Line-end style zero means that the end of the line has a square edge. Line-end style one puts an arrow at the end of the line. Line-end style two means that the end of the line is rounded. These are shown with three thickened lines in Figure 4-5.

0 - Squared (default)

1 - Arrow

2 - Rounded

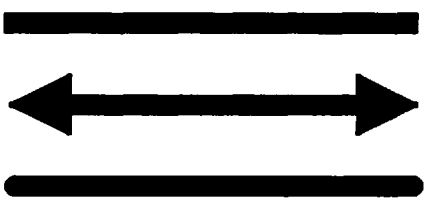


Figure 4-5 Line-End Styles

Function **draw_line( )** again pauses before showing the different marker types. A marker type is a symbol used at a point. This is especially useful when you need to draw several line graphs using the same axes, for example, graphing the growth of product A versus product B versus product C. A unique symbol (marker) is used to denote the points on each of the three curves. A different marker type is used for each line.

The section in **draw_line( )** that shows the different marker types works like the section for the line widths and line styles. Function **v_pmarker( )** takes an array of points and places a marker at each point using the current marker type. Function **draw_line( )** attempts to output 12 different marker types. There are six different marker

types that are guaranteed for all devices. Type one is a dot, type two is a plus sign, type three is an asterisk, type four is a square, type five is an "x", and type six is a diamond (see Figure 4-6). Types seven and above are device-dependent; if they do not exist, marker type three is used as a default.

```
1  .  Dot
2  +  Plus
3  *  Asterisk
4  □  Square
5  X  Diagonal cross
6  ◇  Diamond
7 and up are device dependent
```

Figure 4-6 Marker Types

The markers can be scaled using the **vsm_height( )** function. This sets the height of the marker in terms of y-axis units. If the workstation is using raster coordinates, a y-axis unit is simply a pixel. If on the other hand the workstation uses the normalized device coordinates, a y-axis unit is simply the distance from one discrete point to the next along the y-axis, which may be different than one pixel. All VDI functions that refer to size, height, or width use y-axis or x-axis units. The **vsm_height( )** function cannot be used to scale marker type zero, which is always the smallest dot that VDI can display on the device. After the various marker types are displayed, function **draw_line( )** returns to **main( )**.

Note that most of these output functions also have an associated color attribute, which is ignored for the time being. You can set the color for each output object (for example, line and marker) by using the appropriate set color function. A more detailed discussion of color is handled later in Chapter 6.

Boxed In: Function draw_rect( )

The next function called from main is the **draw_rect( )** function. **Draw_rect( )** uses the **draw_boxes( )** function located directly above **draw_rect( )** in the application function section of GRAFDEMO.

The **draw_boxes( )** function clears the screen and draws two rows of four rectangles. The four rectangles are drawn using the four different rectangle output functions supplied by VDI: **vr_recfl( )**, **v_rbox( )**, **v_rfbox( )**, and **v_bar( )**. Function **vr_recfl( )** draws a filled rectangle. The **v_rbox( )** function draws a rounded rectangle. The function **v_rfbox( )** produces a rounded, filled rectangle. Finally, func-

tion **v_bar( )** draws another filled rectangle. Why not simply use **vr_recfl( )** to produce a filled rectangle? This is because **v_bar( )**, designed to produce the bars in a bar graph, uses the perimeter setting. The function **vr_recfl( )** does not draw a perimeter regardless of the perimeter attribute setting. Each of these functions has a parameter that is an array of four elements, which defines the two opposite corners of the rectangle to be drawn as in program LINES.

Function **draw_boxes( )** first clears the workstation and calls the function **vsf_perimeter( )**. This function works with most shape functions (circles, ellipses, rectangles) to turn on or off the outline (perimeter) attribute. Outlined figures have a black line drawn around their perimeters (only visible when the figures are filled with a color other than black). For the first row of four boxes, the perimeter attribute is off so these boxes are not outlined. For the second row, the perimeter attribute is on so these boxes are outlined.

The first time **draw_boxes( )** is called from **draw_rect( )**, the attribute settings are still set from the call to **draw_line( )**. Primarily, the line width last set has a value of 9. Note that the perimeter drawn uses the current line-width setting. After the first call to **draw_boxes( )**, function **draw_rect( )** resets the line-drawing attributes to their default values. A common programming mistake is assuming that attributes somehow reset themselves within a program; they don't!

Function **draw_boxes( )** is called four more times to show the various *interior-fill* settings using the **vsf_interior( )** function. The four interior-fill settings are 0 for hollow (no filling), 1 for solid (using the current fill color), 2 for pattern fill, and 3 for hatch fill. There is a fifth interior-fill setting using value 4, which allows the programmer to specify a pattern as the fill pattern. As before, two sets of boxes are drawn each time **draw_boxes( )** is called: the top line with the perimeter attribute turned off and the second line with the perimeter attribute turned on.

Note that the box produced by function **vr_box( )** always has a perimeter drawn independent of the value of the perimeter attribute; this is just the opposite of **vr_recfl( )**. These two functions ignore the current perimeter setting. This was not readily apparent before using different fill styles.

The next section of **draw_rect( )** displays the available fill patterns. First the workstation is cleared and the interior fill style is set to style 2, pattern fill. Even though the program tries to display 32 different patterns on the screen; however, after 24 defined patterns, the remaining patterns default to pattern 1 (see Figure 4-7). They are numbered from 1 to a device-dependent value. For the Atari ST screen, only 24 fill patterns are available.

Function **draw_rect( )** next displays the fill hatches. Again 32 hatches are tried, but only 12 different hatches are produced (see

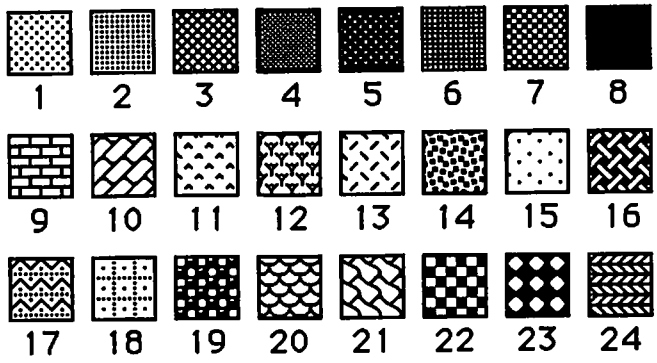


Figure 4-7 Fill Patterns

Figure 4-8). Hatches are also numbered from 1 to a device-dependent maximum. For the Atari ST screen, only 12 hatches are defined.

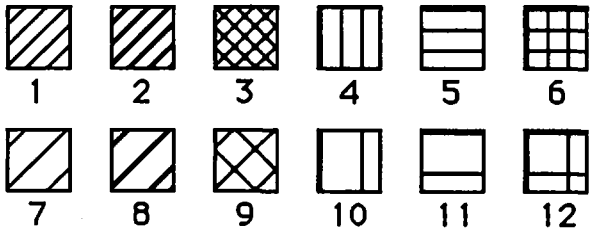


Figure 4-8 Fill Hatches

The VDI provides several functions to fill complex shapes, which are demonstrated next in **draw_rect( )**. First, a bow-tie shape is drawn and filled. The workstation is cleared; the perimeter is turned on so that you can see what is being filled; the interior fill is set to a pattern fill; and pattern style is set to the brick pattern. The function **v_fillarea( )** takes an array of points that define a polygon when connected. The function starts at the first point, draws a line to the next point, and so on until the last point is drawn. When VDI reaches the last point in the array, it automatically connects the first point to the last point so that a closed polygon is created. Then the polygon is filled with the current fill settings (see Figure 4-9).

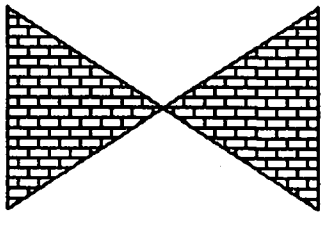


Figure 4-9 Filled Bow Tie

The final section of **draw_rect( )** performs a contour fill. The contour fill function starts at a point on the display surface and continues to fill the bounded area surrounding that point. It's almost like pouring cement into a mold—the area inside the mold (inside the bounded area) is completely filled with the current fill setting. This section first clears the workstation and sets the hollow fill so that the borders (the mold) can be drawn. The writing mode is set to transparent so that overlapping figures do not erase each other. Next, function **v_bar( )** is used to draw two slightly overlapping rectangles.

To demonstrate the contour fill, the interior fill style is set to a hatch fill, and the hatch style is set to 3. The contour fill is done by calling **v_contourfill( )** specifying a point inside the area you want filled (in this case, a point in the middle of the intersection of the two rectangles). This function continues to fill until it reaches a point having the color specified by the last parameter in the function call. Thus, you can fill to the boundaries of a specific color. If the value −1 is used, the filling stops when a color is reached that is different from the color at the starting point. Note that if the area to be filled is not completely bounded, the fill color "spills out" until a boundary (or the edge of the screen) is reached. After the contour fill, **draw_rect( )** returns to **main( )**.

Going in Circles: Function draw_circ( )

The next segment of GRAFDEMO deals with drawing circles, ellipses, arcs, and pie shapes of circles and ellipses. Function **draw_circ( )** first draws a circle and an ellipse. As always, the workstation is cleared before drawing commences. The interior-fill style is set to a pattern fill and the pattern style is selected as a "wavy" pattern. The functions **v_circle( )** and **v_ellipse( )** draw the circle and ellipse. The two parameters following the screen handle in the function calls define the x and y coordinates of the center of the figures. The last parameter of **v_circle( )** (the number 40 in this program) determines the circle's radius. The radius is measured in x coordinate units. Likewise, the last two parameters of **v_ellipse( )** determine the radius in the x and y direction, respectively. The ellipse is calculated around those maximum values.

Next a circle arc and an ellipse arc are drawn. The parameters of **v_arc( )** are the same as those for **v_circle( )**, with the addition of two more values representing the start angle and end angle of the arc. The angle values are measured in tenths of a degree. Thus, the values 800 and 1800 cause an arc to be drawn from 80 degrees to 180 degrees. Zero degrees represents "east" or 3 o'clock, 90 degrees is "north" or 12 o'clock, and so on. The reason that angles are given in tenths of degrees is simple: time. Tenths of degrees can be represented

by integers and still provide a relatively good amount of accuracy. To have the same accuracy while measuring in degrees requires floating point numbers. It takes a computer much longer to compute and draw angles based on floating decimal point numbers than on integers. With integers the angles can still be represented with a fair degree of accuracy and the calculations can be performed much faster. The elliptical arc function **v_ellarc( )** also operates in much the same way as **v_ellipse( )**; again the last two parameters represent the starting and ending angles of the arc.

The last part of **draw_circ( )** draws circle and ellipse pie shapes. The function **v_pieslice( )** works the same as the **v_arc( )** function except that it fills in the pie. Likewise, **v_ellpie( )** works similarly to **v_ellarc( )** and also fills in the pie. Note that the elliptical pie is drawn from 350 degrees to 50 degrees and goes through 0 degrees. Arcs and pies are always drawn counterclockwise; a pie drawn with a starting angle of 1 degree and an ending angle of 359 degrees is an almost complete circle. After you enter program GRAFDEMO, try changing the starting and ending angles.

Type Casting: Function **draw_text( )**

The next and final application function, **draw_text( )**, demonstrates the two VDI text output functions **v_gtext( )** and **v_justified( )**. The function **v_gtext( )** simply outputs a string at the specified point. Justified text is created by **v_justified( )**. Justified text has just enough space between each word and/or letter to make the string a particular length. In this way, a paragraph can have a flush right margin. Both these functions provide their output while in the VDI *graphics* mode, which means that the text is literally drawn on the screen. Note that graphic text is conceptually different from text produced using the **printf( )** function of C or any of the BIOS functions. Graphic text is drawn using the current screen and text attributes. Therefore, you can change text color, writing mode, drawing angle, size, and alignment.

The screen is cleared first and **v_gtext( )** is called to write a string. The two integer parameters are the x and y coordinates of the starting point of the text. The **v_justified( )** function also specifies a starting point, a character string, and the length (in x-axis units) into which the string must fit. The last two parameters determine how the text is to be justified. The first parameter specifies that VDI changes the spacing between *words* to justify the text. If this parameter is FALSE, VDI is not allowed to change interword spacing. The second parameter determines whether VDI changes the spacing between *characters* to justify the text. If it is FALSE, VDI does not change intercharacter spacing. The first call to **v_justified( )** has both these parameters set

to FALSE; therefore VDI is not allowed to justify the text at all. The second call to **v_justified( )** allows the VDI to change intercharacter spacing. Since a space is also considered to be a character, the spaces around each space are enlarged as well as the spaces between each character. The third call to **v_justified( )** allows the VDI to change interword but not intercharacter spacing. The last call to **v_justified( )** in this section allows VDI to change both types of spacing.

The section uses **vst_rotation( )** to set the rotation angle for any graphics text output. The angle is specified in tenths of a degree so the first call rotates text 90 degrees (not 900). Zero degrees represents level text. The function **v_gtext( )** is used to output the text. It prints the text at the angle specified by **vst_rotation( )**. The next call to **vst_rotation( )** rotates the text to 180 degrees (upside down); finally, the rotation is set back to zero. Note that not all devices can rotate text. In fact, one of the many pieces of information returned by **vq_extend( )** is whether the workstation has text rotation capability and whether text may be rotated at any angle or only in 90-degree increments. The Atari ST screen allows only 90-degree increments.

The next section shows the current settings used by the text font. The workstation is cleared and **vqt_attributes( )** is called. Note that functions beginning with "vq" are query functions; functions beginning with "vs" are "set" functions; and functions beginning with just "v" are output functions. The **vqt_attributes( )** function returns information about the current text attributes. These attributes include the current text face, text color, angle of rotation, horizontal and vertical alignment (which is soon discussed), the writing mode, character width and height, and character cell width and height. Here just the current text face and height are shown by using **v_gtext( )** to draw a string.

The next section of **draw_text( )** demonstrates the difference between the various height settings. The height of the character is measured in printer's *points*. One point equals 1/72 of an inch and measures from the base line of one line of text to the base line of the next line of text. The function **vst_point( )** sets the point size to the first parameter after the screen handle. The remaining parameters are values returned by VDI that contain the character width and height and cell width and height of the setting made by **vst_point( )**. If VDI cannot find the specified character height, it automatically picks the largest possible size smaller than the requested size. The same parameters are used by **vst_height( )**, which sets the character height in y-axis units. The first calls in **draw_text( )** set the height to one pixel and one point. You can see that this size really doesn't look like much. The rest of this section tries different point and pixel sizes; default, 10 pixels, 40 pixels, and 72 pixels. Note that 40 point, 40 pixels, 72 point, and 72 pixels all appear to be the same size. Be-

cause these sizes don't exist, the VDI chooses the largest type size it can find.

Font variations are explored next. If you don't have any fonts loaded on your ST, you only see a 6-by-6 system font as the output of this section. The screen is cleared and a loop is used to look at the different fonts. Inside the loop, the **vst_font( )** function sets the font style. The font index is set to the present value of the loop index. The inquiry function, **vqt_name( )**, retrieves the name and style of the font. This function returns a 32-character string. The first 16 characters of the string contain the font name (or the face name as the VDI refers to it). The next 16 characters contain the font style. When this selection has finished playing with the various fonts, **draw_text( )** returns to system font using the value in **attrib[7]** obtained from the **vqt_attributes( )** function called earlier.

Table 4-1: Face Names and Styles

Face Names	*Styles*
Swiss 721	Light
Swiss 721	Thin italic
Dutch 801	Roman
Dutch 801	Bold italic

GEM can access a wide variety of fonts if they are available on your system disk and are available to the device you are using. The ASSIGN.SYS file mentioned in Chapter 1 tells GEM which font files are available to the device. Before your program can use the fonts, the font file must be loaded into the workstation you are using through the **vst_loadfonts( )** function.

To continue with **draw_text( )**, the next section adjusts the text alignment. Alignment can be performed on any of the lines specified by the character cell. Vertical text alignment is the relationship between the y value of the point at which text is drawn and the vertical placement of the text. Vertical text alignment can be performed on the base line (standard), half line, ascent line, bottom line, descent line, or top line. For example, the first call to **vst_alignment( )** causes the base line of the text to be aligned on y value 20. The next call to **vst_alignment( )** causes the half line of the text to be aligned on y value 20.

Horizontal text alignment can also be specified using **vst_alignment( )**. Horizontal alignment is the relationship between the x coordinate of

the point at which text is drawn and the actual horizontal position of the text. For example, horizontal left alignment will align the furthest left point of the string with the x coordinate of the point specified. Center alignment centers the string at the x coordinate; right alignment places the furthest right point of the string at the x coordinate.

The first parameter for **vst_alignment( )** after the screen handle determines horizontal alignment (0 meaning left, 1 meaning center, and 2 meaning right). The next parameter is vertical alignment with the value 0 for base line, 1 for half line, 2 for ascent line, 3 for bottom line, 4 for descent line, and 5 for top line. The different values for vertical alignment are shown first in the output of GRAFDEMO. Note that a left horizontal alignment (the default) is maintained throughout the various vertical changes and that the vertical alignment is not changed by new values for the horizontal alignment. The last two parameters of **vst_alignment( )** simply return the new horizontal and vertical alignment values selected, respectively.

The final workstation attributes shown in **draw_text( )** are the text effects available on the ST: thickened, light intensity, italicized, underlined, outlined, shadowed, or any combination. The **vst_effects( )** function sets the text effect to be used for all subsequent graphic text output based upon the value of the parameter supplied. Each bit in that value determines whether a particular attribute is set on or off. As shown in Table 4-2, bit 0 corresponds to thickened text on or off; bit 1 determines light intensity; bit 2 determines italicization; bit 3 determines underlined text; and so. If the bit has a 0 value, that effect is not used. If the bit has a 1 value, that effect is active.

Table 4-2: Bit Values for Text Effects

Bit	*Description*
0	Thicken
1	Light intensity
2	Skew
3	Underlined
4	Outlined
5	Shadow

Since each bit can be set independently, you can combine various effects. Bits are numbered from right to left; if you want thickened text only, the value of the effect parameter is 1. If you want underlined and thickened text, bits 0 and 3 have to be on, giving the parameter value of 9 (see Figure 4-10).

Normal
Thickened
Light Intensity
Skewed
Underlined
Outlined
Shadowed

Figure 4-10 Text Effects

There are a total of 64 possible combinations. Depending upon the screen and fonts you have, you may not see all the effects. For example, all effects except shadowing are implemented if you use the system font on a monochrome monitor.

Changing GRAFDEMO

This completes the GRAFDEMO program. If you have not already done so, enter this program and run it. Once you have it running, make any changes in it you desire. The program has been modularly designed so that you can change and execute only those areas you want to focus on. For example, try changing the line drawing functions. All you need to do is to call **draw_line( )** from **main( )** so you can comment out the other function calls.

Now is an appropriate time to experiment with parameter values, become familiar with the VDI functions and how they work, and learn how to reference the appendices of this book. Some changes that you can make to GRAFDEMO are playing with text point and height values and changing **calc_shape( )** to produce different shapes to see

how the array is filled and used. (If you try changing **calc_shape( )**, make sure you return the proper number of points; otherwise VDI gets confused and draws either too few or too many points.)

Designing Your Own Patterns

Giving the user the ability to specify a line style provides greater variety to the output of your programs. Designing your own line or fill patterns is much like filling in a small bit map. For a line, the pattern consists of a linear set of 16 bits contained within one word on the ST. Wherever a bit is set, a dot is said to be at that point on the line. If the bit is zero, no dot is at that point on the line. Of course, the actual output depends on the current writing mode. Figure 4-4 shows the default line styles available on the ST.

Designer Lines

To use your own line style, you must set the line style index to 7 (the user-defined line style). The actual line style is set using the function **vsl_udsty( )**, the VDI Set Line User-Defined Style. Its parameters are the screen handle followed by a 16-bit value (declared as WORD on the Atari), which sets the bits for the line style.

Look at program USERTYPE (see Listing 4-2). Function **main( )** shows that this program performs two basic tasks: setting a user-defined line style and setting a user-defined fill pattern. Under the application functions section, there is a function called **set_line( )**. This function clears the workstation and sets the line type to the user-defined style (style 7). The variable style (declared as WORD) is set to hold the line style that will be the user-defined line style for **vsl_udsty( )**. Hexadecimal notation is used to set the variable style because it is easier to convert from binary bits to hexadecimal than to decimal. For example, the first line pattern used in **set_line( )** is a series of alternating on and off bits. This translates to a binary value of 1010101010101010. This 16-bit pattern can be broken up into four pieces of four bits each, that is, 1010 1010 1010 1010. Each four-bit segment can represent 16 different values; hence the hexadecimal (base 16) system. The four-bit pattern represents the hexadecimal digit A. Therefore, the 16-bit pattern is equivalent to AAAA base 16. Go through the other three line styles listed in **set_line( )** and practice converting from binary to hexadecimal. This skill is quite useful later on in this book.

After each call to **vsl_udsty( )** with new line style, the function **draw_lines( )** is called. This function draws a series of lines based on

Listing 4-2 Program USERTYPE

```
/*********************************************************************
     USERTYPE.C     User-defined styles program

     This program shows the use of user-defined line styles
     and user-defined fill patterns.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard ID */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int WORD;                /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

WORD pfill_pat[64] = {
     0xFF00, 0xFF00, 0xFF00, 0xFF00,    /* plane 1 */
     0xFF00, 0xFF00, 0xFF00, 0xFF00,
     0x00FF, 0x00FF, 0x00FF, 0x00FF,
     0x00FF, 0x00FF, 0x00FF, 0x00FF,

     0x00FF, 0x00FF, 0x00FF, 0x00FF,    /* plane 2 */
     0x00FF, 0x00FF, 0x00FF, 0x00FF,
     0xFF00, 0xFF00, 0xFF00, 0xFF00,
     0xFF00, 0xFF00, 0xFF00, 0xFF00,
```

Listing 4-2 (continued)

```
     0xFF00, 0xFF00, 0xFF00, 0xFF00,    /* plane 3 */
     0xFF00, 0xFF00, 0xFF00, 0xFF00,
     0x00FF, 0x00FF, 0x00FF, 0x00FF,
     0x00FF, 0x00FF, 0x00FF, 0x00FF,

     0x00FF, 0x00FF, 0x00FF, 0x00FF,    /* plans 4 */
     0x00FF, 0x00FF, 0x00FF, 0x00FF,
     0xFF00, 0xFF00, 0xFF00, 0xFF00,
     0xFF00, 0xFF00, 0xFF00, 0xFF00
};

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
         work_out[57],
         new_handle;                        /* handle of workstation */
int      i;

         for (i = 0; i < 10; i++)           /* set for default values */
              work_in[i] = 1;
         work_in[10] = 2;                   /* use raster coords */
         new_handle = phys_handle;          /* use currently open wkstation */
         v_opnvwk(work_in, &new_handle, work_out);
         return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}
```

Listing 4-2 (continued)

```
/*************************************************
    Application Functions
*************************************************/

draw_lines(x,y)
int x,y;
/*************************************************
Function: Draw an array of lines using the current line style.
Input:    x = x coord to start array.
          y = y coord to start array.
Output:   None.
*************************************************/
{
WORD pxy[4];
int i;

/* draw 8 horizontal lines */
     pxy[0] = x;       pxy[1] = y;
     pxy[2] = x + 50; pxy[3] = y;
     for (i = 0; i < 8; i++)
     {
          v_pline(screen_vhandle, 2, pxy);
          pxy[0]++;                /* move start point right by 1 */
          pxy[1] += 3;             /* move start point down by 3 */
          pxy[2]++;                /* move end points by same */
          pxy[3] = pxy[1];
     }

/* draw 8 vertical lines */
     pxy[0] = x;       pxy[1] = y + 40;
     pxy[2] = pxy[0]; pxy[3] = y + 80;
     for (i = 0; i < 8; i++)
     {
          v_pline(screen_vhandle, 2, pxy);
          pxy[0] += 3;             /* move start point right by 3 */
          pxy[1]++;                /* move start point down by 1 */
          pxy[2] = pxy[0];         /* move end points by same */
          pxy[3]++;
     }

/* draw 8 diagonal lines */
     pxy[0] = x + 60;  pxy[1] = y;
     pxy[2] = x + 110; pxy[3] = y;
     for (i = 0; i < 8; i++)
     {
          v_pline(screen_vhandle, 2, pxy);
          pxy[0]++;                /* move start point right by 1 */
          pxy[1] += 3;             /* move start point down by 3 */
          pxy[2]++;                /* move end right by 1 */
          pxy[3] += 10;            /* move end point down by 10 */
```

Listing 4-2 (continued)

```
    }

    return;
}


set_line()
/*************************************************
Function: Sets user defined line styles and draws them.
Input:    None.
Output:   None.
*************************************************/
{
WORD style;


    v_clrwk(screen_vhandle);
    vsl_type(screen_vhandle, 7);  /* use user-defined line styles */

    style = 0xAAAA;               /* 1010101010101010 */
    vsl_udsty(screen_vhandle, style);
    draw_lines(10,10);

    style = 0xF00F;               /* 1111000000001111 */
    vsl_udsty(screen_vhandle, style);
    draw_lines(10,110);

    style = 0x8CEF;               /* 1000110011101111 */
    vsl_udsty(screen_vhandle, style);
    draw_lines(170,10);

    style = 0xC0F0;               /* 1100000011110000 */
    vsl_udsty(screen_vhandle, style);
    draw_lines(170,110);

    return;
}


set_pattern()
/*************************************************
Function: Sets a user-defined pattern and draws it.
Input:    None. Uses array pfill_pat.
Output:   None.
*************************************************/
{
WORD pxy[4];

    v_clrwk(screen_vhandle);
    vsf_interior(screen_vhandle, 4);   /* use user pattern to fill */
```

Listing 4-2 (continued)

```
                                        /* use 1 plane */
     vsf_udpat(screen_vhandle, pfill_pat, 1);
     pxy[0] = 10; pxy[1] = 10;
     pxy[2] = 80; pxy[3] = 190;
     vr_recfl(screen_vhandle, pxy);

                                        /* use 2 planes */
     vsf_udpat(screen_vhandle, pfill_pat, 2);
     pxy[0] = 90;  pxy[1] = 10;
     pxy[2] = 160; pxy[3] = 190;
     vr_recfl(screen_vhandle, pxy);

                                        /* use 4 planes */
     vsf_udpat(screen_vhandle, pfill_pat, 4);
     pxy[0] = 170; pxy[1] = 10;
     pxy[2] = 240; pxy[3] = 190;
     vr_recfl(screen_vhandle, pxy);

     return;
}

/*************************************************
     Main Program
*************************************************/

main()
{
int ap_id;                              /* application init verify */

WORD gr_wchar, gr_hchar,                /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();               /* Initialize AES routines */
     if (ap_id < 0)                     /* no calls can be made to AES */
     {                                  /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                     /* set exit value to show error */
     }
     screen_phandle =                   /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();                 /* Get screen attributes */
/*************************************************
     Application Specific Routines
*************************************************/
```

Listing 4-2 (continued)

```
     set_line();
     Crawcin();
     set_pattern();

/*************************************************
     Program Clean-up and Exit
*************************************************/
/* Wait for keyboard before exiting program */
     Crawcin();                    /* GEMDOS character input */
     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
/*************************************************/
```

the given coordinates. Function **draw_lines( )** outputs eight horizontal lines, eight vertical lines, and eight diagonal lines, with each line in each set offset slightly from the previous line. When you look at the output, note that the horizontal lines don't start consistently. Look at the set of horizontal lines in the lower left section of the output, for instance. The line style is set to F00F in this function call (four ones, eight zeroes, four ones). The first dash in each line is not the same length; yet the remaining blanks and dashes are all lined up. This does not happen with the vertical or diagonal lines because the VDI tries to be extra efficient by copying the entire line style word for horizontal line. Since every 16 bits in one line on the screen represents one word in memory, a new word in memory starts every 16 pixels. Because the line style is stored in a word, the VDI copies the

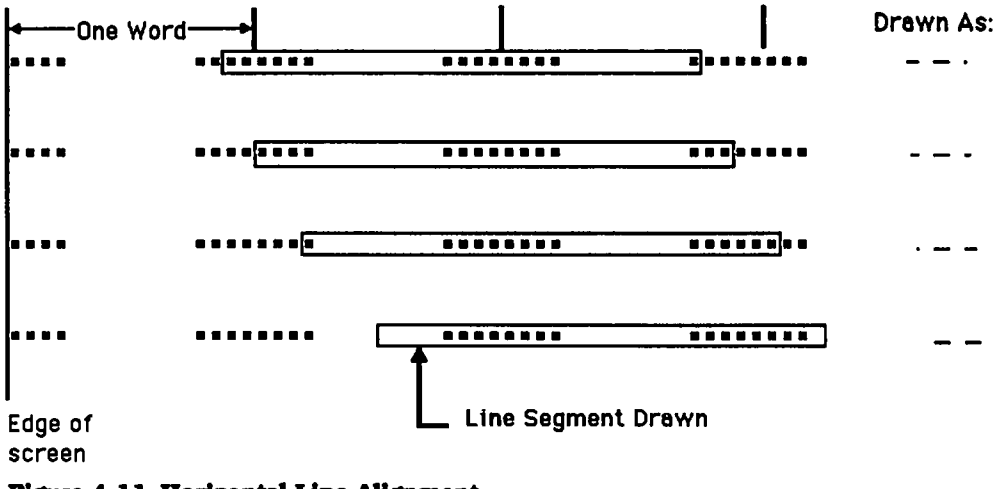


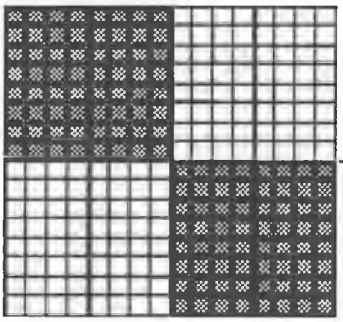
Figure 4-11 Horizontal Line Alignment

line style word to the screen words that contain the line. Any points beyond the ends of the line are not changed. Figure 4-11 shows an example of a line drawn on the screen. The brackets in the figure show the start and endpoints of the line. Note that the line style is *word-aligned*, which means that the start of the line style always corresponds with the start of a memory word. Word alignment is also applied to the system line styles when drawn horizontally.

Having word-aligned horizontal lines poses a problem. If the line has spaces in it, it may not appear to start where you expect. As an example, look at the last line in Figure 4-11. The line should start at pixel 23, but when it is drawn it appears to start at pixel 28. Nonhorizontal lines do not have this problem because the VDI must copy the line bit for bit to the output.

Finding a Pattern

Lines by themselves are fairly straightforward. You simply put down a set of bits on the screen to form them. Patterns, on the other hand, are somewhat more complicated. A pattern consists of sixteen 16-bit words giving you a 16-by-16 matrix of bits (see Figure 4-12).



```
FF00
FF00
FF00
FF00
FF00
FF00
FF00
FF00
00FF
00FF
00FF
00FF
00FF
00FF
00FF
00FF
```

Figure 4-12 Checkerboard Pattern

The pattern in Figure 4-12 forms a checkerboard. The hexadecimal code for this pattern is contained in the array **pfill_pat** listed in the application-specific data section of USERTYPE. Each word (that is, each element) represents one line of the pattern. Note that there are four *planes* listed. Each plane is a fully defined pattern. Planes are a concept used with color output and are discussed further when color

is covered. For now, program USERTYPE just demonstrates the fact that the user-defined fill pattern has the capacity to account for color output as well as monochrome output. Those of you using a monochrome monitor see no difference. Those of you using a color monitor may wish to return to this program after color output is discussed and see how the planes affect the output.

The routine **set_pattern( )** draws the pattern three times, each time using a different number of planes. Function **set_pattern( )** starts by clearing the screen and setting the interior fill style to the user-defined fill pattern. The function **vsf_udpat( )** sets the user-defined pattern to the pattern held in **pfill_pat( )**. The last parameter of **usf_udpat( )** indicates the number of planes to use. When USERTYPE draws the pattern, the pattern at the edges of the rectangle is not complete. Just as with horizontal lines, all patterns drawn by the VDI are word-aligned. For the most part, word alignment doesn't cause any problems here as it does for horizontal lines.

Changing USERTYPE

Essentially that's it for the user-defined lines and patterns. You may want to try changing the user-defined pattern. Try to make the square of the checkerboard smaller or create something completely different. You may also want to try emulating one of the predefined patterns. Patterns can be a lot of fun to play with!

Multiple Workstations

Multiple workstations, the last part of this chapter, complete the discussion of attributes and VDI output. You have already seen that a workstation is a mechanism the VDI uses to output graphic images. A workstation has a set of attributes associated with it. Look in Appendix A for the functions **v_opnwk( )** and **v_opnvwk( )**, which open physical and virtual workstations, respectively. Also look at the workstation inquiry function **vq_extnd( )** used in the GEM-related function **set_screen_attr( )** of the programs. Take particular notice of the values returned by this function.

The open workstation functions returns "a slew" of information about the workstation: the maximum addressable width in the x and y directions; the width and height of the pixel, the number of line types and widths available; the number of marker types and sizes; the number of faces supported; the number of patterns; the number of hatches; the number of predefined colors; color capability; text rotation; fill area capability; input device capability; character width and

height; maximum line width; and maximum and minimum marker heights. The extended inquiry function **vq_extnd( )** has two options: it can return the same values as **v_opnwk( )** or it can give extended information such as the type of screen being used, the number of background colors, whether text effects are supported, the number of planes associated with the color, character rotation capability, and the number of writing modes available.

The program MULWORK (see Listing 4-3) demonstrates the use of multiple workstations. An example of the use of multiple workstations given in Chapter 3 is the graphic design program in which each type of output is given its own workstation.

Listing 4-3 Program MULWORK

```
/*********************************************************************
     MULWORK.C      Multiple workstations

     This program demonstrates the use of multiple workstations
     to implement several sets of drawing attributes on the same
     physical device. Also this program introduces the use of
     the clipping rectangle.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM RES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int WORD;                /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */
```

Listing 4-3 (continued)

```
/*************************************************
     Application Specific Data
*************************************************/

WORD      handle1,                /* handles for virtual wkstations */
          handle2;

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
    work_in[10] = 2;                     /* use rastar coorcs */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/
```

Listing 4-3 (continued)

```
set1_attrib()
/*************************************************
Function: Set attributes for virtual workstation 1.
Input:    None. handle1 must be set.
Output:   None. Sets attributes.
*************************************************/
{
     vswr_mode(handle1, MD_REPLACE);     /* replace writing mode */
     vsl_type(handle1, 3);               /* dotted lines */
     vst_effect(handle1, 8);             /* underlined text */
     return;
}


set2_attrib()
/*************************************************
Function: Set attributes for virtual workstation 2.
Input:    None. handle2 must be set.
Output:   None. Sets attributes.
*************************************************/
{
     vswr_mode(handle2, MD_TRANS);       /* transparent writing mode */
     vsl_width(handle2, 5);              /* use thicker lines */
     vst_rotation(handle2, 2700);        /* text at 270 degrees */
     vsf_interior(handle2, 2);           /* use pattern fill */
     vsf_style(handle2, 5);              /* set pattern to use */
     return;
}


draw_rect()
/*************************************************
Function: Draw rectangles on both workstations.
Input:    None. handle1 and handle2 must be set.
Output:   None.
*************************************************/
{
WORD pxy[4];

     pxy[0] = 10; pxy[1] = 10;
     pxy[2] = 75; pxy[3] = 125;
     vr_recfl(handle1, pxy);        /* draw filled rectangle */


     pxy[0] = 100; pxy[1] = 10;
     pxy[2] = 165; pxy[3] = 125;
     vr_recfl(handle2, pxy);

     return;
}
```

Listing 4-3 (continued)

```
draw_text()
/*************************************************
Function: Draw text on both workstations.
Input:    None. handle1 and handle2 must be set.
Output:   None.
*************************************************/
{
     v_gtext(handle1, 50, 50, 'This text is drawn on workstation 1.');
     v_gtext(handle2, 120, 70, 'This text is drawn on workstation 2.');
     return;
}




clip_demo()
/*************************************************
Function: Demonstrate clipping rectangles.
Input:    None. handle1 and handle2 must be set.
Output:   None.
*************************************************/
{
WORD pxy[4];

     pxy[0] = 20;  pxy[1] = 20;
     pxy[2] = 100; pxy[3] = 100;
     vs_clip(handle1, TRUE, pxy);  /* set clipping rect on 1 */

     pxy[0] = 100; pxy[1] = 20;
     pxy[2] = 180; pxy[3] = 100;
     vs_clip(handle2, TRUE, pxy);  /* set clipping rect on 2 */

     v_clrwk(screen_vhandle);      /* clear screen */
     pxy[0] = 100; pxy[1] = 20;
     pxy[2] = 100; pxy[3] = 100;
     v_pline(screen_vhandle, 2, pxy);   /* draw line to show edge */

     pxy[0] = 80;  pxy[1] = 30;
     pxy[2] = 150; pxy[3] = 50;
     vr_recfl(handle1, pxy);       /* draw rect on 1 over clip edge */

     pxy[0] = 50;  pxy[1] = 60;
     pxy[2] = 120; pxy[3] = 80;
     vr_recfl(handle2, pxy);       /* draw rect on 2 over clip edge */

                                   /* draw circle */
     v_circle(handle2, 180, 120, 50);
     return;
}
```

Listing 4-3 (continued)

```
/*************************************************
     Main Program
*************************************************/


main()
{
int ap_id;                          /* application init verify */


WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;


WORD work_in[11], work_out[57];
/*************************************************
     Initialize GEM Access
*************************************************/


     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws('***> Initialization Error. <***\n');
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }
     screen_phandle =               /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Get screen attributes */
/*************************************************
     Application Specific Routines
*************************************************/


                                    /* open two virtual workstations */
     handle1 = open_vwork(screen_phandle);
     handle2 = open_vwork(screen_phandle);

     set1_attrib();                 /* set attributes for each */
     set2_attrib();

     draw_rect();                   /* draw rectangles in both */
     draw_text();                   /* draw text in both */
     Crawcin();                     /* pause */

     clip_demo();                   /* show clipping rectangle */
     Crawcin();
```

Listing 4-3 (continued)

```
     v_clsvwk(handle2);              /* close all workstations */
     v_clsvwk(handle1);
/*************************************************
     Program Clean-up and Exit
*************************************************/
/* Wait for keyboard before exiting program */
     Crawcin();                      /* GEMDOS character input */
     v_clsvwk(screen_vhandle);       /* close workstation */
     appl_exit();                    /* end program */
}
/*************************************************/
/**/
```

MULWORK also shows the use of the *clipping rectangle*. The clipping rectangle is a device used by graphic programs to prevent drawing outside a specific area. In other words, the clipping rectangle clips (ignores) any output that goes beyond the region of the rectangle. The clipping rectangle itself is not visible on the screen. It's analogous to looking out a window: you can't see through the walls, but you can see through the window. Each workstation can have its own clipping rectangle. One use for the clipping rectangle is when the graphic design program is made. The workstation for the drawing area has a clipping rectangle that encloses just the work area. In this way, any output written outside the work area is not written into its neighboring areas. The same goes for error messages and settings areas.

In the MULWORK program, the only application-specific data used are the variables to hold the handles of the two workstations manipulated throughout the program. Look at **main( )**. The general flow of MULWORK first opens two workstations, sets the attributes for each workstation, and puts output to both workstations. Finally, the clipping rectangle is demonstrated.

The application-specific routines begin by opening two virtual workstations using the **open_vwork( )** function. This function can be called for as many virtual workstations as the VDI allows, depending on the version of the VDI, the type of machine used, and the amount of memory available. Usually four or eight virtual workstations can be open simultaneously. In MULWORK, there are three virtual workstations and a physical workstation open.

Once the workstations are open, the first thing to do is to set the attributes of each workstation using the routines **set1_attrib( )** and **set2_attrib( )**. Workstation one uses the replace writing mode. The width of the lines used is default (one), the line type is a dotted line, and the text effect is underlined text. Workstation two uses a transparent writing mode. The line width is set slightly thicker (five),

the text is rotated at 270 degrees, and the interior is filled with fill pattern five.

Functions **draw_rect( )** and **draw_text( )** draw rectangles and text on both workstations, respectively. The output on workstation one uses a solid fill (the default); the text is drawn right-side up and underlined; and whatever is drawn replaces what was on the screen. The output on workstation two, a lattice fill pattern, appears to be laid over what was on the screen before it. That's why it is called the "transparent" writing mode. The text, drawn at 270 degrees, is also transparent and so the background shows through blank areas of the character cells. The text is not underlined on this workstation.

After the pause in **main( )**, the **clip_demo( )** routine is called to create the clipping rectangles. The use of a clipping rectangle can be turned on or off. Off is the default value when a workstation is opened. Setting a clipping rectangle is done in much the same way as drawing a rectangle. The array **pxy** contains the points that define two opposite corners of the rectangle. The function **vs_clip( )** is called to set the clipping rectangle. The first parameter to **vs_clip( )** is the workstation handle. The second parameter turns clipping on or off with TRUE on and FALSE off. The third parameter defines the clipping rectangle itself. Function **vs_clip( )** is called twice to create a clipping rectangle on each workstation. The two clipping rectangles lie next to one another but on different workstations. The screen is cleared and a line is drawn on the edge between the two clipping rectangles so that you can see the boundary. This line is drawn on the default screen virtual workstation so that it is not affected by the clipping rectangles.

With two adjacent clipping rectangles on the screen, **clip_demo( )** tries to draw beyond the boundaries of each rectangle. First, it draws a rectangle on workstation one that goes beyond the right edge of the clipping rectangle. The program's output shows that the rectangle stops right at the boundary. Another rectangle is drawn on workstation two so that it exceeds the left edge. Again the output shows that the rectangle stops right at the boundary. Finally, a circle is drawn on workstation two with an origin completely outside of the workstation's boundary. However, its radius causes some of the circle to fall within the clipping rectangle. Only that portion of the circle falling within the clipping rectangle is shown.

MULWORK ends by closing all virtual workstations used by the program. Try creating some functions that draw lines, text, or arcs, for example, to experiment with the attributes on different workstations. Go into the set attributes function to change some of the attributes and see how they affect the output.

CHAPTER FIVE

Treasure Maps

This chapter discusses bit maps and their use in displaying information on the screen. A bit map is an extremely important concept. You have already seen what a bit map does and basically how it works. It should be stressed that a bit map is used primarily for screen output. The Atari machine sets aside 32,000 bytes of memory for screen use. The hardware that produces the screen images looks at this memory block to produce the images shown on the screen.

Implementing a Bit Map

Computer memory on most machines such as the Atari is broken down into units called *bytes*. A byte consists of eight bits with each bit set to a value of zero or one. The central processing unit of most computers can access memory as a single byte, a word, or some larger unit. On the Atari ST, a *word* consists of two bytes with the second byte following the first in memory. A larger unit that might be available is called a *page*. On the ST, a page is a block of 512 bytes. Each byte in a computer's memory has an *address*, which is a number ranging from zero to the total number of bytes available in the system. When accessing a particular byte, your program may specify any address within this range. When accessing a particular word, the address must be an even value. Thus, the first word in memory consists of the two bytes at addresses 0 and 1. The second word in memory is the two bytes at addresses 2 and 3. When you program in a language like C, the compiler handles the restrictions

for accessing bytes and words in most cases. However, when working with the ST's graphics display, you need to account for certain memory restrictions, for example, the location of the bit map when it is accessed by the display hardware.

A final aspect of the bit map that you need to know is how bits are numbered. In a byte, the furthest right bit is called bit number 0 and the furthest left bit is bit 7. In a word, the two bytes are taken to be one long string of bits, so the furthest right bit is bit 0 and the furthest left bit is bit 15.

The Bit Map in Memory

The graphic subsystem of the ST allocates 32,000 contiguous bytes in memory for the bit map. For each of the 32,000 bytes, one byte follows the next with no spaces in between. Figure 5-1 shows that the 32,000-byte screen memory block can reside anywhere in memory. For processing efficiency, the address of the first word in this block must lie on a *half-page boundary*. This is an address evenly divisible by 256. Because the central processing unit on the ST is more efficient when processing words than bytes, all accesses to the bit map are in terms of words.

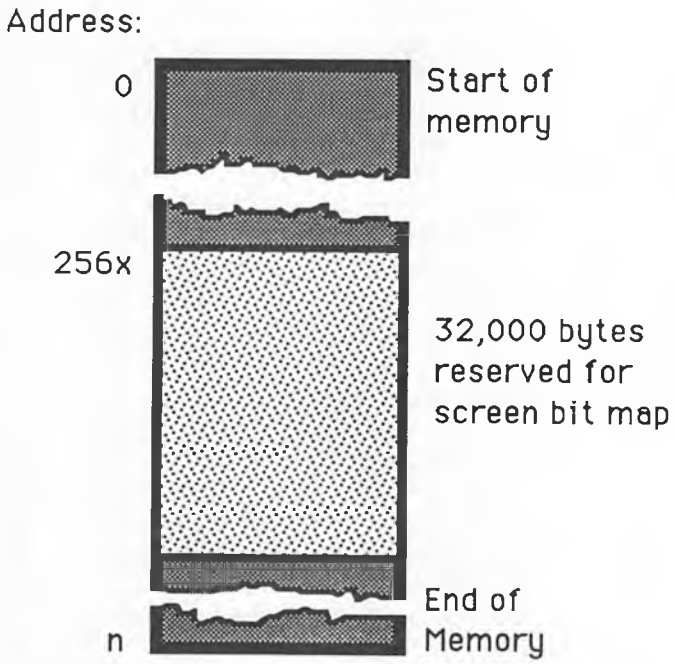


Figure 5-1 Bit Map in Random-Access Memory

When the graphics hardware reads this section of memory, each word is mapped to the screen. Each pixel on the screen is represented by one of the bits in the bit map. The value of the bit is either 1 or 0, which determines whether the pixel is "on" or "off." Figure 5-2 demonstrates how this is done.

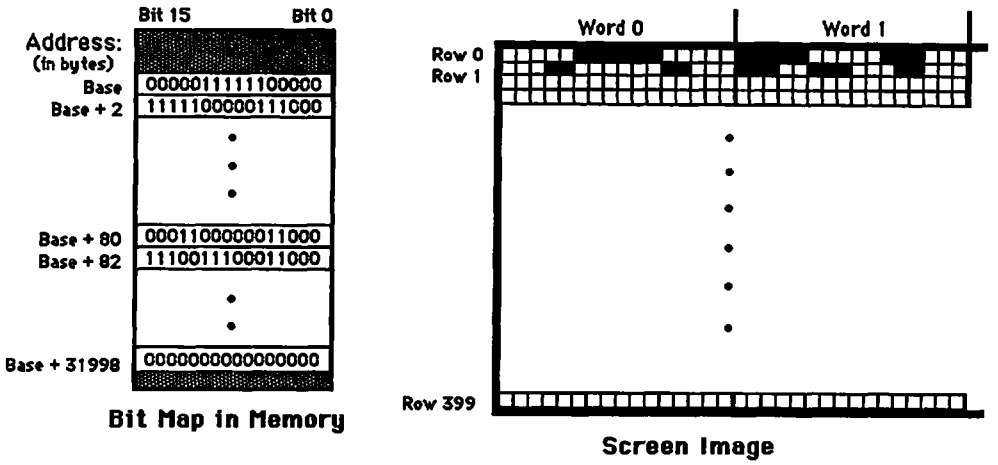


Figure 5-2 Mapping of Memory to Screen in Monochrome Mode

The beginning of a bit map has an address called the *base address*, noted by the name *Base* in Figure 5-2. The next word is at address *base* plus two (two bytes per word); the word after that is at address *Base* plus 4; and so on. A video display has a fixed number of rows. A monochrome monitor has 400 rows, numbered zero through 399. Within each row is a set number of pixels. In the case of a monochrome monitor, there are 640 pixels represented in 40 words (16 bits per word times 40 words equals 640 bits or pixels). Each word in a row is numbered from zero through 40. Thus, the screen is laid out as a matrix where each pixel has a y coordinate (the row it is in) and an x coordinate (the word and then particular bit within that word).

However, memory is not stored as a matrix. Memory is stored in a linear manner with one byte following the next from start to finish. To transfer the data in memory into an image on the screen, the graphics hardware is designed to *map* the linear representation in memory onto the two-dimensional matrix of the screen. The mapping process starts at the first address in the bit map and the upper left corner of the screen. The first word in the bit map is placed onto the first 16 bits in the first row; each word contains 16 bits. The next word is placed in the next 16 bits and so on. This process continues from left to right until the end of the row is reached; this is

represented by the word with the address *Base* plus 39. Then the graphics hardware moves down one row to the next line on the screen. This line is filled using the next 40 words in memory again going from left to right. The mapping process continues until all the lines have been mapped (representing 40 words per row times 400 rows, or 32,000 bytes).

Chapter 1 shows how the screen works. The inside of the screen is coated with a phosphor that glows momentarily when struck by an electron. The time required for the glow to fade depends upon the type of phosphor used but is generally about 1/30 of a second. Therefore, the entire screen must be rewritten, or *refreshed*, before the pixels fade. If the pixels fade and are then redrawn, the screen appears to flicker.

The time between the start of one refresh cycle and the start of the next cycle is called the *refresh rate*. The refresh rate on most screens is 1/30 of a second (30 Hz). Thus, the graphics hardware must map the entire memory to the screen within this time so that it can start the process over. The Atari color monitors work at this rate. The Atari monochrome monitor works at a slightly faster rate (35.6 Hz). The refresh rate is significant in animation programs because you need to know how much time there is between one frame and the next.

Note that the process of drawing on the screen actually consists of changing the value of a bit within the bit map. Therefore, if you set a bit to 1, the next time the graphics hardware maps that bit to the screen that pixel is displayed. To the user it looks as though the dot is drawn on the screen.

Mapping the Bits

Each time your program draws a pixel on the screen, the graphics routine in the computer performs a calculation to convert an (x,y) coordinate into a memory address and a bit number within a word. This occurs for *every* pixel your program draws, such as when a circle or rectangle is drawn, a figure is filled, and a single pixel is drawn.

Given a coordinate pair (with x ranging from 0 to 639 and y ranging from 0 to 399 for the monochrome screen), the address that contains this pixel equals the following: the base address of the bit map **plus** y times 40 (to account for the number of words in the previous rows) **plus** the integer value of x divided by 16 (since x represents the pixel position). This is shown in Equation 1. To find the exact bit that represents the (x,y) pair, take the remainder of x divided by 16 and subtract this value from 15 as shown in Equation 2. This subtraction is used because bits are numbered from right to left and from 0

through 15. This is the operation performed by the graphics hardware each time a point is drawn on the screen.

Equation 1 address = base + (y * 40) + int(x / 16)
Equation 2 bit = 15 − (x / 16)

Program BITMAP

Look at the program BITMAP shown in Listing 5-1. This program shows how you can manipulate the screen bit map itself. Screen bit map manipulation must be done through the extended BIOS (XBIOS) routines. The XBIOS is used because the bit map is a hardware-dependent feature and not accessible through GEM.

There are two addresses used by the display hardware. One is the *physical* base address, which is the address used by the display hardware when it needs to read from the bit map to put data onto the display. The other is the *logical* base address, which is used when the VDI (or some part of your software) is writing to the bit map. Program BITMAP uses both of these addresses.

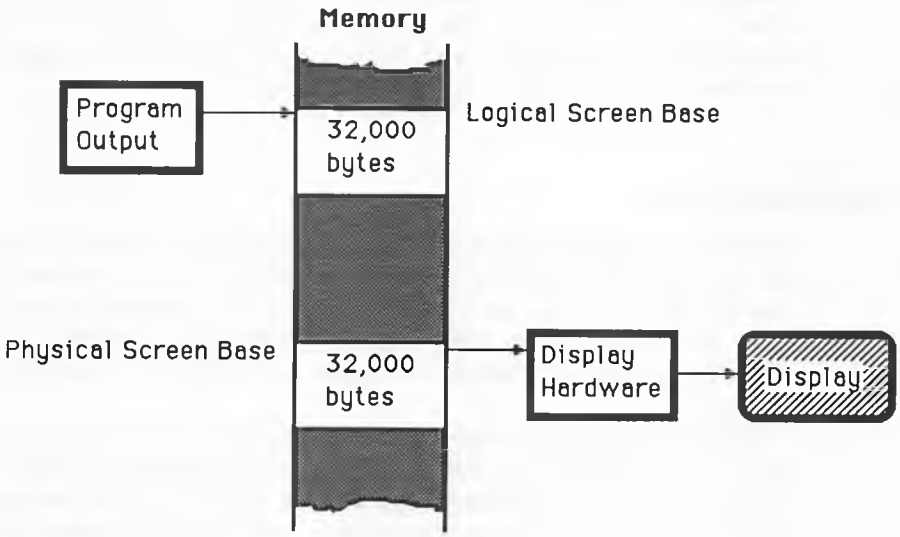


Figure 5-3 Use of Logical and Physical Screen Base Addresses

Figure 5-3 shows what happens in program BITMAP. Program output is placed into memory wherever the *logical* screen address is pointing. The display hardware outputs the bit map pointed to by the

physical screen base address. By taking advantage of this situation, a program can draw a complex image in the logical bit map. When the image is complete, the physical address can be set to point to the logical bit map, causing it to be displayed on the screen. The end result is that the user perceives an instantaneous image.

In the program under the application-specific data, there are three character pointers. The first, **old_bitmap**, contains the logical base address of the screen currently being used. The variable **new_bitmap** contains the address of a second bit map to be set up within the program. Finally, the variable **old_pbase** contains the physical base address of the screen currently being used.

In main( ), functions **set_base( )**, **test1( )**, and **test2( )**, demonstrate some of the uses and variations of the screen bit map. The next function, **Mfree( )**, is like the **free( )** function in C except that it is defined under GEMDOS. The GEMDOS function is preferred because it is consistent with the Atari machine and any other hardware environments that use GEM.

Listing 5-1 Program BITMAP

```
/*********************************************************************
     BITMAP.C Demonstrate bitmaps

     This program shows how a bitmap is used for screen display.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include <stdio.h>                  /* Standard IO */
#include <osbind.h>                 /* GEMDOS routines */
#include <gemdefs.h>                /* GEM AES */
#include <obdefs.h>                 /* GEM constants */

#define  FALSE     0
#define  TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef    int WORD;                /* WORD is 16 bits */
WORD       contrl[12],              /* VDI control array */
           intout[128], intin[128], /* VDI input arrays */
           ptsin[128], ptsout[128]; /* VDI output arrays */
```

Listing 5-1 (continued)

```
WORD        screen_vhandle,          /* virtual screen workstation */
            screen_phandle,          /* physical screen workstation */
            screen_rez,              /* screen resolution 0,1, or 2 */
            color_screen,            /* flag if color monitor */
            x_max,                   /* max x screen coord */
            y_max;                   /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

char *old_bitmap,                    /* logical base of current screen */
     *new_bitmap,                    /* logical base of new screen */
     *old_pbase;                     /* physical base of curr. screen */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
```

Listing 5-1 (continued)

```
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/

set_base()
/*************************************************
Function: Allocate memory for new screen bitmap.
Input:    None.
Output:   Sets old_bitmap, new_bitmap, and old_pbase.
*************************************************/
{
#define  BOUNDARY 256
long x;

/* allocate new screen bitmap */
     x = (long)Malloc(32256L);           /* get 32 kbytes */
     if (!(x % BOUNDARY))                /* on half page boundary */
          new_bitmap = (char *) x;
     else                                /* move to boundary */
          new_bitmap = (char *)(x + (BOUNDARY - (x % BOUNDARY)));

/* get current values */
     old_bitmap     = (char *)Logbase();
     old_pbase      = (char *)Physbase();
     return;
}

test1()
/*************************************************
Function: Show access to second bitmap.
Input:    None.
Output:   None.
*************************************************/
{
int i;

     printf("\n\nlogical = %lx,\nphysical = %lx,\nnew = %lx\n",
          old_bitmap, old_pbase, new_bitmap);
     printf("\nPress any key to see new bitmap\n");
     Crawcin();
     Setscreen(new_bitmap, new_bitmap, -1);
     for (i = 0; i < 32767; i++)
          *(new_bitmap+i) = i;
     Crawcin();
     Setscreen(old_bitmap, old_bitmap, -1);
     return;
}
```

Listing 5-1 (continued)

```
test2()
/*************************************************
Function: Show hidden drawing on second bitmap.
Input:    None.
Output:   None.
*************************************************/
{
WORD pxarray[4];
int  i;

     printf("\n\nClearing new bitmap\n");
     Setscreen(new_bitmap, -1L, -1);
     v_clrwk(screen_vhandle);
     Setscreen(old_bitmap, -1L, -1);
     printf("\nPress any key to see\n");
     Crawcin();
     Setscreen(-1L, new_bitmap, -1);
     Crawcin();
     Setscreen(old_bitmap, old_bitmap, -1);

     printf("\nDrawing in new bitmap\n");
     Crawcin();
     Setscreen(new_bitmap, -1L, -1);
     for (i = 10; i <= 150; i += 15)
     {
          pxarray[0] = i;     pxarray[1] = i;
          pxarray[2] = i+10;  pxarray[3] = i+10;
          vr_recfl(screen_vhandle, pxarray);
     }
     Setscreen(old_bitmap, -1L, -1);
     printf("\nNew screen ready\n");
     Crawcin();
     Setscreen(new_bitmap, new_bitmap, -1);
     Crawcin();
     Setscreen(old_bitmap, old_bitmap, -1);
     return;
}

/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/
```

Listing 5-1 (continued)

```
     ap_id = appl_init();          /* Initialize AES routines */
     if (ap_id < 0)                /* no calls can be made to AES */
     {                             /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                /* set exit value to show error */
     }

     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */

/**************************************************
     Application Specific Routines
**************************************************/

     v_curhome(screen_vhandle);
     set_base();
     test1();
     test2();

/**************************************************
     Program Clean-up and Exit
**************************************************/

     Mfree(new_bitmap);            /* return memory */

/* Wait for keyboard before exiting program */
     Crawcin();                    /* GEMDOS character input */
     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */

}
/**************************************************/
```

Allocating a Bit Map

The routine **set_base( )** allocates 32,000 bytes of memory for a new, secondary screen bit map. There is no input to this routine and the only output is that the function sets the values of variables **old_bitmap**, **new_bitmap**, and **old_pbase** mentioned above. The routine first allocates 32,000 bytes of memory using the **Malloc( )** function. Note that the function name is capitalized; this means that it is a GEMDOS function and not a C-defined function. **Malloc( )** returns a value of type *long* (representing an address in memory) indicated by our (redundant) *long* type cast operator. On the Atari machine, longs and pointers are both 32 bits wide. A pointer is not

used because GEM does not know what type of pointer is required. Therefore, a type cast is probably necessary in any case. Note that the argument used in the call to **Malloc( )** is 32,256, even though the program only needs 32,000 bytes. The reason is that the base address of the screen bit map must be on a half-page boundary (divisible by 256). By adding on an extra 256 bytes, the program has enough room to shift the base address upward so that it is divisible by 256 and still has 32,000 bytes of memory allocated. Thus, after allocating the memory space, **set_base( )** tests the address of the memory block to see if it falls on a half-page boundary. If it does, **new_bitmap** is set to that address; otherwise, **new_bitmap** is set to the next highest half-page boundary. The **set_base( )** routine next sets the variable **old_bitmap** to the value of the logical base address returned by the **Logbase( )** function and then sets the variable **old_pbase** to the value of the physical base address returned by **Physbase( )**. The function **Logbase( )** and **Physbase( )** are XBIOS functions because they are hardware-dependent values set by the ST when it is turned on. Function **set_base( )** has now set all the base addresses needed for this program.

Using the New Bit Map

The routine **test1( )** demonstrates how to access a second bit map. Function **test1( )** first prints the values of the logical and physical bit map base addresses. After the user presses a key, function **Setscreen( )** is called. This function has three arguments. The first is the address of the logical screen bit map; the second is the address of the physical screen bit map; and the third is a value to set the screen resolution. This last value is analogous to the value returned by **Getrez( )** in routine **set_screen_attr( )** where 0 indicates low resolution, 1 indicates medium resolution, and 2 indicates high resolution. Any of the arguments of **Setscreen( )** may be negative values. The use of a negative value indicates that no action is to be taken on that parameter. For example, the first call to **Setscreen( )** sets the logical and physical screen base addresses but does not change the screen resolution. The end result of function **Setscreen( )** changes the logical address, the physical address, or the screen resolution. When this occurs, the new bit map will be displayed on the screen.

For demonstration purposes, **test1( )** performs a loop that simply puts a number (the value of the index) into all locations of the bit map one byte at a time. Because the program directly affects the screen's bit map, the result of this loop appears on the screen. Note how the screen reflects the binary values of each location. For example, in row 0 byte 0 (set to 0 in the loop) contains eight white pixels. The next byte (set to a value of one) contains seven white pixels

and one black pixel. The next bytes continue this pattern. To end the test, **Setscreen( )** is called a second time to restore the old screen bit map, which remains unaffected by the manipulations with the new bit map.

The next routine, **test2( )**, shows how to set the logical and physical bit maps independently. This is done so that you can have one image displayed while your program is drawing another image on the second bit map. The image displayed is in the bit map pointed to by the physical base address; the image drawn is pointed to by the logical base address.

The first step in **test2( )** is to tell the user that the other bit map is being cleared. Function **Setscreen( )** is called to set the logical base address of the bit map to **new_bitmap**, but no changes are made to the physical base or the resolution. The VDI function **v_clrwk( )** is used to clear the bit map. Note that in drawing the VDI uses the logical base address, not the physical base address. Thus, the physical bit map (the one displayed to the user) shows no change. This follows the division of purpose for each bit map where the logical bit map holds the output from the software and the physical bit map produces the image on the screen. When the second bit map is cleared, you see no effect on the first bit map (the one displayed). After the bit map is cleared, function **Setscreen( )** is called to restore the logical base address so the output of the print statement appears on the screen. After the user presses a key, the physical base address is set without changing the logical base address or the resolution. This shows the newly cleared bit map. After another key is pressed, **text2( )** returns the display to the original bit map **old_bitmap**.

Note that in the calls to **Setscreen( )**, a constant of type *long* is used for the values of −1. Because **Setscreen( )** expects an address, the parameter *must* fill all 32 bits of information. If an *int* was used, only 16 bits of data would be passed, causing the parameters to be read incorrectly.

The next section of **test2( )** draws in the new bit map while the old bit map is displayed. The image drawn is a set of filled rectangles along a diagonal. While the drawing takes place, the user sees no changes to the screen. Once the boxes have been drawn, the user presses a key to see them. When another key is pressed, the original bit map is displayed again.

Note that all **printf( )** calls start and end with the newline character. For some reason, **printf( )**, a C output function, does not work precisely with changes made by **Setscreen( )**. If there is no newline before text is printed out, the text may either appear on the new or the old bit map.

As an operational side note, the logical screen base address is set immediately by **Setscreen( )** because your program wants to output to

the new bit map setting as soon as possible. The physical base address, however, is set only when the screen has finished a refresh cycle (that is, when the last word of the screen has been drawn). During the time between refresh cycles, the physical base address is set. If this were not the case and the physical base address was set in the middle of the refresh cycle, the top half of the screen would show the old bit map and the bottom half of the screen the new bit map.

Program BITMAP shows that you can set and use a logical bit map while looking at a physical bit map. This is a very important feature in animation. Try changing the defined boundary value of 256 in **set_base( )** to a value of 127. This causes some rather interesting results when the bit map is changed from the old bit map to the new bit map.

Program ANIMATE

Program ANIMATE is an application of the techniques learned in program BITMAP. Starting with the application-specific data in the program listing (see Listing 5-2), the constant SQSIZE determines the size of the squares that are drawn. The pointer variables **screen1** and **screen2** hold the address of the two screen bit maps. Variable **screen1** holds the address for the current screen bit map (the one used by the system) and **screen2** is the bit map created for the program.

Listing 5-2 Program ANIMATE

```
/*********************************************************************
     ANIMATE.C Demonstrate animation techniques

     This program shows how a two bitmaps are used for
     animation.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include <stdio.h>                  /* Standard IO */
#include <osbind.h>                 /* GEMDOS routines */
#include <gemdefs.h>                /* GEM AES
#include <obdefs.h>                 /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/
```

Listing 5-2 (continued)

```
/* Declare global arrays for VDI. */
typedef   int  WORD;                /* WORD is 16 bits */
WORD      contrl[12],               /* VDI control array */
          intout[128], intin[128],  /* VDI input arrays */
          ptsin[128], ptsout[128];  /* VDI output arrays */

WORD      screen_vhandle,           /* virtual screen workstation */
          screen_phandle,           /* physical screen workstation */
          screen_rez,               /* screen resolution 0,1, or 2 */
          color_screen,             /* flag if color monitor */
          x_max,                    /* max x screen coord */
          y_max;                    /* max y screen coord */

/*************************************************
     Application-Specific Data
*************************************************/
#define   SQSIZE     10             /* size of a square */

char *screen1,                      /* logical base of current screen */
     *screen2;                      /* logical base of new screen */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                   /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)           /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                   /* use raster coords */
     new_handle = phys_handle;          /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];
```

Listing 5-2 (continued)

```
     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/

set_base()
/*************************************************
Function: Allocate memory for new screen bitmap.
Input:    None.
Output:   Sets screen1, screen2.
*************************************************/
{
#define BOUNDARY
long x;

/* allocate new screen bitmap */
     x = (long)Malloc(32256L);          /* get 32 kbytes */
     if (!(x % BOUNDARY))               /* on half page boundary */
          screen2 = (char *) x;
     else                               /* move to boundary */
          screen2 = (char *)(x + (BOUNDARY - (x % BOUNDARY)));

/* get current screen */
     screen1   = (char *)Logbase();
     return;
}

draw_box(a1, a2)
WORD a1[], a2[];
/*************************************************
Function: Draw two vertical lines of boxes.
Input:    a1 = array for first line of boxes.
          a2 = array for second line of boxes.
Output:   None.
*************************************************/
{
          for (a1[1] = a2[1] = 10, a1[3] = a2[3] = a1[1] + SQSIZE;
               a1[1] < y_max - 20;
               a1[1]+=30, a2[1]+=30, a1[3]+=30, a2[3]+=30)
          {
               vr_recfl(screen_vhandle, a1);
               vr_recfl(screen_vhandle, a2);
          }
          return;
}
```

Listing 5-2 (continued)

```
animate1()
/*************************************************
Function: Use single bitmap to animate boxes
Input:    None.
Output:   None.
*************************************************/
{
WORD p11[4], p12[4];                /* squares x coord */

     p11[0] = SQSIZE * 2;           /* square 1 on screen 1 start */
     p11[2] = p11[0] + SQSIZE;

     p12[0] = x_max - (SQSIZE * 2);      /* square 2 on screen 1 start */
     p12[2] = p12[0] + SQSIZE;

     v_clrwk(screen_vhandle);       /* clear screen */
                                    /* set XOR drawing mode */
     vswr_mode(screen_vhandle, MD_XOR);
     draw_box(p11, p12);       /* draw initial squares */

     while (p12[0] > SQSIZE+5)
     {
          draw_box(p11, p12);
          p11[0]++; p11[2]++;                /* move along x values */
          p12[0]--; p12[2]--;
          draw_box(p11, p12);
          Vsync();
     }
}
animate2()
/*************************************************
Function: Use multiple bitmaps to animate boxes
Input:    None. Uses screen1 and screen2.
Output:   None.
*************************************************/
{
WORD p11[4], p12[4],                /* screen 1 squares x coord */

     p21[4], p22[4];                /* screen 2 squares x coord */

     p11[0] = SQSIZE * 2;           /* square 1 on screen 1 start */
     p11[2] = p11[0] + SQSIZE;

     p12[0] = x_max - (SQSIZE * 2);      /* square 2 on screen 1 start */
     p12[2] = p12[0] + SQSIZE;

     p21[0] = (SQSIZE * 2 + 1);          /* square 1 on screen 2 start */
     p21[2] = p21[0] + SQSIZE;

     p22[0] = x_max - (SQSIZE * 2 + 1); /* square 2 on screen 2 start */
     p22[2] = p22[0] + SQSIZE;
```

Listing 5-2 (continued)

```
     Setscreen(screen1, screen1, -1);
     v_clrwk(screen_vhandle);       /* clear screen */
     Setscreen(screen2, screen2, -1);
     v_clrwk(screen_vhandle);       /* clear screen */
                                    /* set XOR drawing mode */
     vswr_mode(screen_vhandle, MD_XOR);
                                    /* draw initial squares */
     Setscreen(screen1, -1L, -1);
     draw_box(p11, p12);
     Setscreen(screen2, -1L, -1);
     draw_box(p21, p22);

     while (p12[0] > SQSIZE+5)
     {                              /* show new and draw on old */
          Setscreen(screen1, screen2, -1);
          Vsync();
          draw_box(p11, p12);           /* erase */
          p11[0]+=2; p11[2]+=2;         /* move along x axis */
          p12[0]-=2; p12[2]-=2;
          draw_box(p11, p12);           /* draw */
                                    /* show old and draw on new */
          Setscreen(screen2, screen1, -1);
          Vsync();
          draw_box(p21, p22);           /* erase */
          p21[0]+=2; p21[2]+=2;         /* move along x axis */
          p22[0]-=2; p22[2]-=2;
          draw_box(p21, p22);           /* draw */
     }
     Setscreen(screen1, screen1, -1);
}

/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
```

Listing 5-2 (continued)

```
        Crawcin();
        exit(-1);                   /* set exit value to show error */
    }

    screen_phandle =                /* Get handle for screen */
        graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
    screen_vhandle = open_vwork(screen_phandle);
    set_screen_attr();              /* Get screen attributes */

/**************************************************
    Application Specific Routines
**************************************************/

    set_base();
    animate1();
    Crawcin();
    animate2();

/**************************************************
    Program Clean-up and Exit
**************************************************/

    Mfree(screen2);                 /* return memory */

/* Wait for keyboard before exiting program */
    Crawcin();                      /* GEMDOS character input */
    v_clevwk(screen_vhandle);       /* close workstation */
    appl_exit();                    /* end program */

}
/**************************************************/
/***********/
```

In animation speed is the key. All changes made to the screen images must be made fast enough that the eye does not detect individual images. This is the principle behind television and motion pictures. For example, motion pictures are composed of frames that contain still images. When they are projected fast enough (about 24 frames per second), the *illusion* of motion is created. The same is true in computer animation. While one image is displayed, the computer must draw the next image before it is displayed. This represents one of the most difficult aspects of computer animation, especially in applications such as flight simulators where very complex images must be generated within a very brief time.

ANIMATE animates two columns of small boxes that start at each edge of the screen and move past each other to the opposite edge. The program draws each column of boxes at both edges, changes their

horizontal coordinates a small amount, and draws them again repeating this loop until the columns reach the opposite edge. Of course, the previous set of boxes must be erased when the new column is drawn; therefore, ANIMATE uses the XOR writing mode just as program LINES.

In function **main( )**, the first call is to **set_base( )** to set the base values. Function **set_base( )** is the same function used in program BITMAP except that in this program the function sets the variables **screen1** and **screen2**. Next the two animation functions, **animate1( )** and **animate2( )**, are called. The routine **animate1( )** shows the use of a single bit map to animate the boxes. Function **animate2( )** uses two bit maps.

The application function **draw_box( )** is used by both **animate1( )** and **animate2( )** to draw the two vertical columns of boxes. This function takes the coordinates of the top box and draws all the boxes in the column. Function **draw_box( )** draws enough boxes to fill the entire vertical height of the screen as determined by the variable **y_max** set in **set_screen_attr( )**. Having this dynamic drawing routine allows the program to be run on either a monochrome or color monitor.

The function **animate1( )** starts by setting the initial coordinates of box columns one and two in arrays **p11** and **p12**, respectively. Next the workstation is cleared, the writing mode is set to XOR, and the initial set of squares is drawn. In the loop, the current boxes are redrawn first (to erase them). Then the x coordinates for the new columns are set by incrementing the x value of column 1 (which moves from left to right) and decrementing the x value of column 2 (which moves from right to left). When the new x values have been set, the columns are drawn. The loop is terminated when the second column comes within five plus one square width pixels of the left edge of the screen.

The function **Vsync( )** is called to help give the animation a smoother look. The **Vsync( )** function is an XBIOS function and causes the program to pause until the next refresh cycle begins (called a *vertical interrupt*). The **Vsync( )** function allows time for the current refresh cycle to complete so that all the boxes are shown on the screen before they are moved. If you take out the **Vsync( )** function, you see a noticeable flickering effect as the boxes are drawn.

The **animate2( )** function is similar to **animate1( )** except that two bit maps are used. Each bit map has two columns of squares as before. However, instead of redrawing the columns each time on the *same* screen as the columns move from one side to the other, this function displays one bit map while drawing the *next* image on the second bit map. When the second bit map is complete, the purposes of the bit maps are inverted: the second bit map becomes the dis-

played bit map (physical bit map) and the first bit map becomes the drawing bit map (logical bit map). The bit maps keep switching back and forth until the boxes have completed their journey. The boxes appear to move in the same manner as produced by **animate1( )**.

In **animate2( )**, the variables **screen1** and **screen2** each point to the base of a different bit map. The arrays **p11** and **p12** contain the x coordinates of the columns on screen one; the arrays **p21** and **p22** contain the x coordinates of the columns on screen two. As in the previous function, the initial coordinates of the boxes are set. Screen one uses the first position of the boxes while screen two uses the *next* position of the boxes as the starting coordinates. The next position is one pixel closer to the center of the screen. Both screens are cleared, the drawing mode is set to XOR, and the first columns are drawn on each screen.

To start the drawing process, the base addresses need to be set. Since screen one is already visible, the first step in the loop is to switch screens. The logical base address is set to **screen1** and the physical base address is set to **screen2**. Now screen two is visible and screen one is used for graphic output. The **Vsync( )** function pauses the program so that the display hardware has time to show the new image.

While screen two is displayed, **draw_box( )** is called to erase the current columns on screen one and the new positions of these columns are calculated. These new positions are the next position after the image shown on screen two. Therefore, the x coordinates must be moved by *two* and not just one. Function **draw_box( )** is called to draw the new columns on screen one using the coordinates stored in **p11** and **p12**. Now screen one is displayed again and drawing is done on screen two. The procedure for drawing on screen two is exactly the same as the one for drawing on screen one. This flip-flop between display screens continues until the columns have made their trek across the screen. At this point, the screen addresses are set to their original state (that is, the logical and physical base addresses are both set to **screen1** and the function returns to **main( )**.

The last section of **main( )** calls the **Mfree( )** function to free up the memory that was allocated for the second screen (referenced by variable **screen2**). You should always free allocated space or GEM might not be able to use this memory again. Also, be sure to use the corresponding allocation and de-allocation functions (both should be from either C or GEMDOS). The GEMDOS functions are preferred here because of their guaranteed portability.

The output of **animate1( )** shows the boxes moving toward and then past each other. Even though the **Vsync( )** function is in use, the screen still flickers. This is because the user is actually watching the boxes being drawn, erased, and redrawn at each location. Since the

entire process is visible on the screen, a flickering effect occurs. If only one or two boxes are drawn instead of an entire column, the time required to draw and erase the images is reduced to the point where the screen does not flicker. However, if you are creating a game, it is most likely that there are 20 or 30 objects to be moved between images. The more objects to be moved, the more noticeable the drawing procedure becomes.

The output of **animate2( )**, on the other hand, is much smoother and does not flicker. The time required for the boxes to traverse the screen is exactly the same as in **animate1( )**, but **animate2( )** is much more pleasant to view. You may notice that the boxes tend to jump at certain places. This is because the boxes are not drawn fast enough to be placed on the screen during the next available refresh cycle. In other words, sometimes the new bit map has to wait an additional refresh cycle before it can be placed on the screen. The bit maps are always forced to wait for the right time by the **Vsync( )** function.

ANIMATE is quite straightforward. It basically demonstrates the important technique of swapping bit maps. This technique is used later in this book in a more elaborate animation program.

Bit maps provide an extremely powerful tool in the realm of computer graphics. In this chapter, you saw how a bit map is transferred from memory to the screen. You also saw that the display hardware uses two addresses when dealing with a bit map. The logical bit map address points to the base of the bit map to be used for program graphic output. The physical bit map address points to the bit map used by the display hardware when it is transferring an image to the screen. If the logical and physical bit maps point to the same location, the program graphic output is immediately displayed on the screen. If the logical and physical bit map addresses are different, the program can create complex images in the background and have these images displayed instantaneously on the screen.

The bit maps covered in this chapter deal specifically with monochrome bit maps. The introduction of color adds a new dimension to both your program output and the layout of the bit map. Chapter 6 deals with the use of colors and specifically looks at the color implementation on the Atari ST.

CHAPTER SIX

Colors of the Rainbow

This chapter shows how the Atari machine produces color graphics from a bit map, how color is represented in memory and converted to a screen image, and how the VDI and extended BIOS functions alter the color display. This chapter also includes a demonstration program showing some of the capabilities of the color display.

Color Display Implementation

A monochrome screen is coated with a phosphor on the inside that glows when struck with an electron. Because only one type of phosphor is used, only two colors can be displayed: either black (no color) or white/green/amber (depending upon the phosphor used). A color screen uses the three *primary colors* of light, which can be combined to create other colors as described in Chapter 2. On the Atari ST, the intensity of each color can range from 0 (meaning the color is not used, that is, off) to 7 (indicating the highest intensity). This gives eight different intensity levels for each color. Therefore, the Atari ST is capable of displaying 512 different colors: 8 × 8 × 8.

Because there are eight intensity levels for a color, each pixel would require at least three bits to store the intensity level of one color. Since there are three colors with three bits for each color, the computer system would need to store nine bits of information for every pixel on the screen, which would use too much memory. Even at the low resolution setting of 320-by-200 pixels, the system would need 72,000 bytes to store a screen image: 320 pixels × 200 pixels × 9 bits

÷ 8 bits per byte. In other words, the system would need over twice as much memory space to represent one-fourth the resolution of a monochrome screen, which uses only 32,000 bytes in high resolution mode. In addition to this, a group of nine bits cannot be stored efficiently because bytes are only 8 bits long whereas words are 16 bits long.

Monochrome Bit Maps

Atari has devised a mechanism to get around this problem. The monochrome bit map is quite straightforward. Figure 6-1 depicts this.

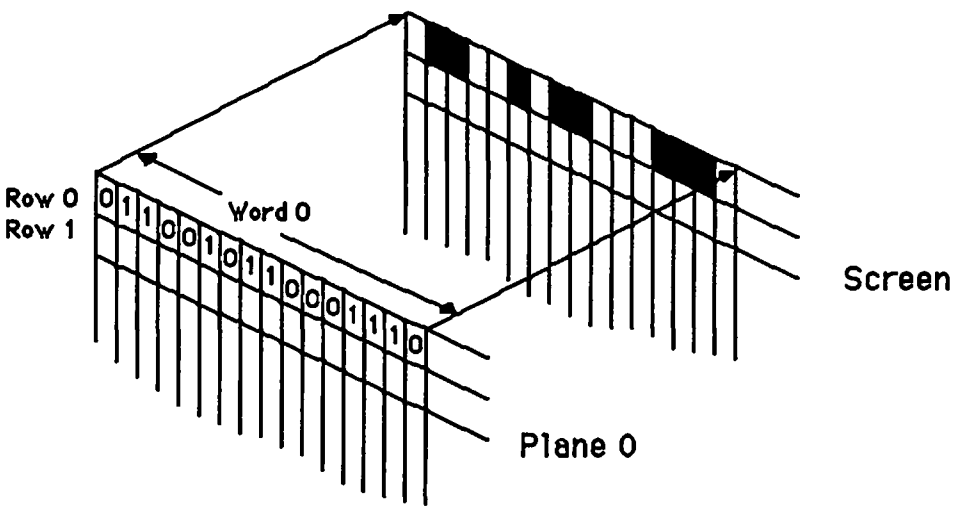


Figure 6-1 Planes Used in Monochrome Bit Map

In high resolution mode, the system requires exactly 32,000 bytes to represent the entire screen in a bit map. The memory allocated for this bit map is shown in Figure 6-2. There is a base address for the start of the bit map and the words within the bit map are contiguous. The first 40 words represent the first row on the screen; the next 40 the second row; and so on.

The Color Palette

On the Atari ST, the color monitor would require over twice the amount of memory needed for the monochrome monitor to represent 512 different colors at one-quarter of the resolution. To keep the bit map for the display within 32,000 bytes, Atari chose a method that allows only 16 colors to be accessible for display at any one time. These 16 colors are placed in what is called a *color palette*.

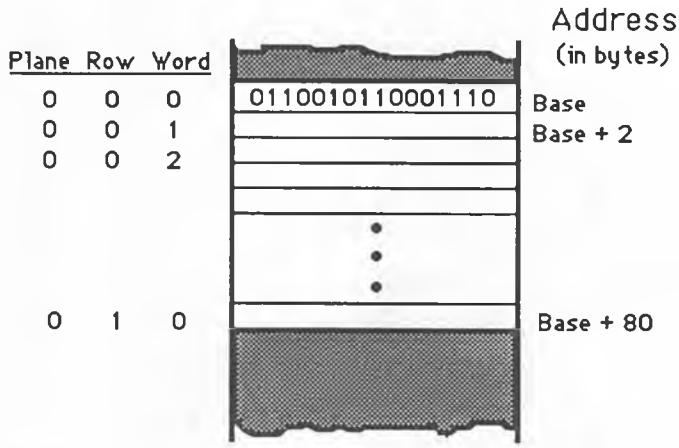


Figure 6-2 Memory Layout of a Monochrome Bit Map

The color palette is a block of 16 words in memory. Each word represents a different color entry. The colors are numbered from 0 to 15. Each entry in the color palette contains the intensity levels for each of the three primary colors used to create this color entry (see Figure 6-3).

In each word, the first four bits represent the blue intensity, the next four bits represent the green intensity, and the next four bits represent the red intensity. Since only three bits are required to describe the eight different levels of intensity, the last bit in each four-bit group is not used. The last four bits of the word are also unused. For example, the color black has a palette entry that is all zeros (0x0000 in hexadecimal). The color white has the full intensity for each of the three colors (0x0777 in hexadecimal).

The color palette shown in Figure 6-3 is actually the default color palette of the author's computer. Colors 0 and 15 should always be set so that color 0 has the highest intensity (white) and color 15 uses no colors (black). The entries in the palette can be modified by a program, which is demonstrated shortly.

How does using a color palette reduce the memory requirements of the bit map? Since the palette has only 16 colors, you don't need 9 bits to represent each pixel on the screen. To represent 16 different colors, you need only 4 bits to represent a number between 0 and 15. On the low resolution screen, the system uses 32,000 bytes: 320 pixels × 200 pixels × 4 bits per pixel ÷ 8 bits per byte. This is the same amount of memory needed for the high resolution monochrome display.

Bit

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Color Index
	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	0
	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	2
	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	3
	0	0	0	0	0	0	0	0	0	1	1	1	0	1	1	4
	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	5
	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	6
	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	7
	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	8
	0	0	0	0	0	1	1	1	0	1	1	1	0	1	1	9
	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	10
	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	11
	0	1	1	1	0	0	0	0	0	1	1	1	0	1	1	12
	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	13
	0	1	0	1	0	0	0	0	0	1	0	1	0	1	0	14
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
	Red				Green				Blue							

Intensities

Figure 6-3 Color Palette

Planes

There are different ways to represent the four-bit color palette index in the bit map. The Atari ST uses a rather interesting method called *planes*.

The color bit map can be thought of as a simple extension of the monochrome bit map shown in Figure 6-1. Since it takes four bits to describe the color of a single pixel, the color bit map uses four planes (see Figure 6-4). Each plane is a bit map representation of the screen. Thus, the furthest left bit in the first byte of each plane corresponds to the upper left pixel on the screen. By combining bit settings in each of the four planes for that particular pixel, a palette entry index can be constructed. For example, in Figure 6-4, the first pixel has a palette index of 0 because bit 15 of the first word in each plane is set to 0. The seventh pixel has a palette entry index of 7 because of the bit settings in each plane.

Similar to the monochrome bit map, the color bit map is laid out in a contiguous block of memory. The first word in each plane describes the first 16 pixels on the screen, the next word in each plane describes the next 16 pixels, and so on. The planes are arranged

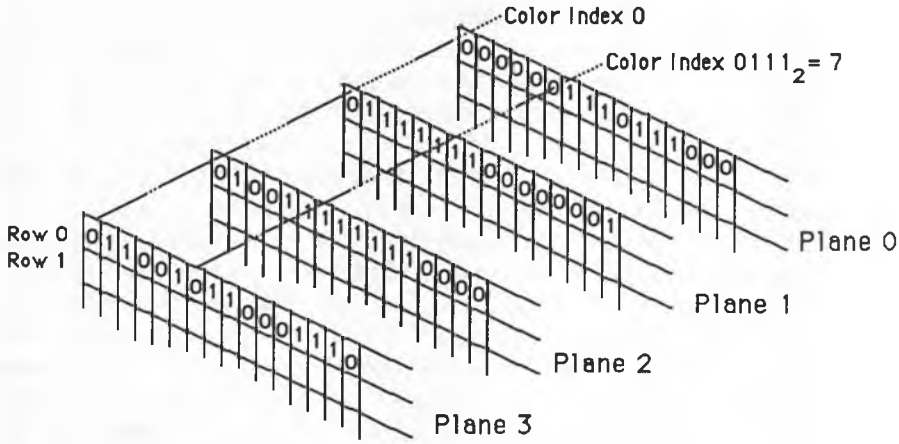


Figure 6-4 Planes Used in a Color Bit Map

in an *interleaved* fashion. This means that the first word of plane 0 is followed by the first word of plane 1, the first word of plane 2, and then the first word of plane 3. The next word in memory corresponds to the second word of plane 0, followed by the second word of plane 1, and so on (see Figure 6-5).

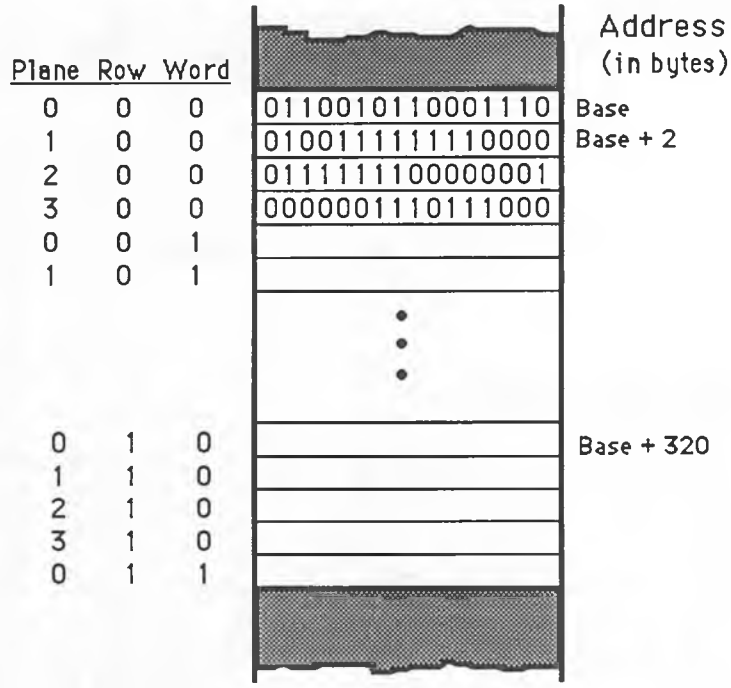


Figure 6-5 Memory Layout of a Color Bit Map

The mapping of bits from a row and word position to pixels on the screen is analogous to the procedure used for a monochrome bit map. It proceeds from left to right on the screen and from top to bottom. The low resolution color screen is 320 pixels wide and has 200 rows. Taking the interleaving of planes into consideration, row 0 occupies the first 160 bytes in the bit map: 20 words per row per plane × 4 planes. Row 1 occupies the next 160 bytes, and so on.

Figure 6-6 shows an example of how color output is produced. First, the appropriate bit is extracted from each plane in the bit map. These bits are placed together to obtain the color palette index. Note that plane 3 is the most significant bit of the color index and plane 0 the least significant bit. In Figure 6-6, the value 0111 references color index 7 in the palette. The value taken from the palette determines the red, green, and blue intensities to be output. This mix of colors is then placed at the corresponding pixel on the screen.

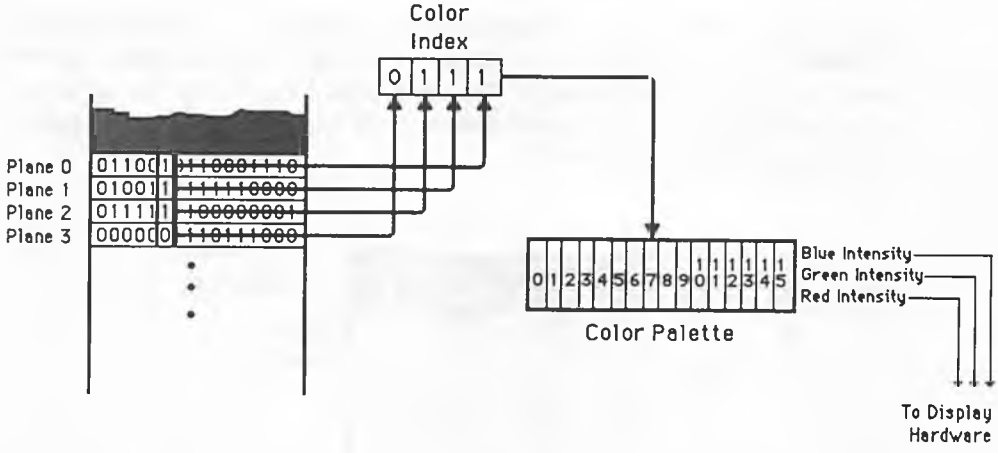


Figure 6-6 Converting Bit Map Planes to Color Output

Color Versus Monochrome

In the programs used so far, there has been no distinction between color and monochrome screens. However, there is a difference. If you try to write color images to a monochrome screen using the VDI, it will map any nonwhite colors as black. Thus, portions of your image using different colors are not distinguishable from one another. This is why it is important to use the inquiry functions as used in **set_screen_attr( )** to determine the type of screen in use. If you are not using the VDI, the value of bit 0 in the color palette entry is used

to determine if the pixel is on or off when output to a monochrome screen.

If you output an image designed for a monochrome monitor to a color screen, the result really won't be too much different except that the colors are white and some other color. For example, if you use colors 0 and 1 the image includes white and whatever color entry 1 is set to be. If you want to be sure black and white are output whether or not you use a color monitor, specify color 0 for white and color 15 for black. As default values, color 0 is the background color (generally white) and color 15 is black. Using this combination you get black-and-white images on all types of monitors. Of course, if your program is using the VDI for output, you may use color 0 and any other color because the VDI converts any nonwhite color to black.

Resolution

Another variable on the video display of the ST is the *resolution* of the image. There are three resolution values: high, medium, and low. In high resolution, the screen has 640 pixels across and 400 pixels down. Medium resolution has 640 pixels across and 200 pixels down, and low resolution has 320 pixels across and 200 pixels down. High resolution is only available on the monochrome monitor for two reasons. First, a color monitor with this resolution is very expensive to produce. Second, the bit map required would consume too much memory. The low resolution mode uses four planes to allow the display of any of 16 colors at one time. Medium resolution has twice the number of pixels as low resolution. However, to fit the bit map into 32,000 bytes, only half the number of planes can be used. This means that only four colors are available in medium resolution. These colors are the first four palette entries.

Program COLOR

Program COLOR is a demonstration program that uses most of the GEM and XBIOS functions for altering the colors displayed on the screen. Listing 6-1 shows the usual header files and GEM application overhead. In the section for application-specific data, there are three constants: RED, GREEN, and BLUE, which are used to reference the array elements of the parameter array for the VDI **set_color( )** function.

Listing 6-1 Program COLOR

```
/*********************************************************************
     COLOR.C  Color demonstration program

     This program shows the use of the color palette.
*********************************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM structures */
#include  <obdefs.h>               /* GEM write modes */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

/* define colors */
#define   RED       0
#define   GREEN     1
#define   BLUE      2

/* rotating palette */
WORD pal_wheel[] = {
     0x000, 0x007, 0x070, 0x700, 0x077, 0x707, 0x770, 0x777,
     0x000, 0x111, 0x222, 0x333, 0x444, 0x555, 0x666, 0x777,
     0x000, 0x007, 0x070, 0x700, 0x077, 0x707, 0x770, 0x777,
     0x000, 0x111, 0x222, 0x333, 0x444, 0x555, 0x666, 0x777
};

/* current palette */
WORD save_pal[16];
```

Listing 6-1 (continued)

```
/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}


set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/

check_table(handle)
WORD handle;
/*************************************************
Function: Check if color lookup table is supported.
Input:    handle    = handle of device to check.
```

Listing 6-1 (continued)

```
Output:   0    = table supported.
          1    = not supported.
*************************************************/
{
WORD work_out[57];

     vq_extnd(handle, 1, work_out);      /* do extended inquire */
     return(work_out[5]);                /* return flag */
}

show_palette()
/*************************************************
Function: Draw a set of squares to show current color settings.
Input:    None.
Output:   None.
*************************************************/
{
register  i;
WORD      pxy[4];

     for (i = 0; i < 16; i++)      /* ST supports up to 16 colors */
     {
          pxy[0] = (i % 4) * 40 + 20;
          pxy[1] = (i / 4) * 40 + 20;
          pxy[2] = pxy[0] + 35;
          pxy[3] = pxy[1] + 35;
          vsf_color(screen_vhandle, i); /* set fill color */
          vr_recfl(screen_vhandle, pxy);
     }
}


change_palette()
/*************************************************
Function: Change palette index 3 using VDI function.
Input:    None.
Output:   None.
*************************************************/
{
WORD rgb[3],                    /* new settings */
     srgb[3];                   /* saved settings */
long delta;                     /* use long to slow down */

/* inquire current value for index 3 */
     vq_color(screen_vhandle, 3, 0, srgb);


/* initialize */
     rgb[RED] = 0;
     rgb[GREEN] = 0;
     rgb[BLUE] = 0;
     delta = 1;
```

Listing 6-1 (continued)

```
/* loop around colors */
     for (rgb[BLUE] = 0; rgb[BLUE] < 1000; rgb[BLUE] += delta)
          vs_color(screen_vhandle, 3, rgb);

     for (rgb[RED] = 0, rgb[GREEN] = 0; rgb[RED] < 1000;
               rgb[RED] += delta, rgb[GREEN] += delta)
          vs_color(screen_vhandle, 3, rgb);

     for (; rgb[BLUE] > 0; rgb[BLUE] -= delta, rgb[GREEN] -= delta)
          vs_color(screen_vhandle, 3, rgb);

     for (; rgb[RED] > 0; rgb[RED] -= delta)
          vs_color(screen_vhandle, 3, rgb);

     Crawcin();
     vs_color(screen_vhandle, 3, srgb); /* restore color */
}

rot_palette()
/************************************************
Function: Use Xbois call to set entire palette.
Input:    None. Uses array pal_wheel[].
Output:   None.
************************************************/
{
register i;

/* save current palette */
     for (i = 0; i < 16; i++)
          save_pal[i] = Setcolor(i, -1);      /* read current value */

     for (i = 0; i < 16; i++)
     {
          Setpalette(&pal_wheel[i]);          /* set base of palette */
          show_palette();
          Crawcin();
     }

     Setpalette(save_pal);                    /* restore original */
}

/************************************************
     Main Program
************************************************/

main()
{
int ap_id;                          /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;
```

Listing 6-1 (continued)

```
/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();          /* Initialize AES routines */
     if (ap_id < 0)                /* no calls can be made to AES */
     {    /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                /* set exit value to show error */
     }

     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */


/*************************************************
     Application Specific Routines
*************************************************/


     v_clrwk(screen_vhandle);

     if (!check_table(screen_vhandle))  /* check for color table */
     {
          printf("***> Color look-up table not supported <***\n");
          printf("Color palette function will have no effect\n");
          printf("Press any key\n");
          Crawcin();
     }

     vsf_interior(screen_vhandle, 1);   /* set solid fill */
     show_palette();                    /* show current palette */
     Crawcin();
     change_palette();                  /* reset values */
     Crawcin();
     rot_palette();


/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     Crawcin();                    /* GEMDOS character input */
     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
/*************************************************/
```

The array **pal_wheel** is used as a set of palette entries. Each line of the initialization contains eight hexadecimal numbers. Each number determines a particular color for that color index. For example, the first entry has a value of zero, which means that the red, green, and blue intensities are all zero and yield the color black. The second entry has a hexadecimal value 0x007, which means that red and green have an intensity of zero and blue has an intensity of 7 (the highest intensity available) producing the color bright blue. The value 0x070 represents green, 0x700 represents red, 0x077 represents a green and blue mixture, and so on. The last entry in the first row, 0x777, represents white. In the next row, the values go through a simple pattern from 0x000, 0x111, 0x222, to 0x777; this represents eight different shades of gray from black to white. The last two lines repeat the first two lines.

The last application-specific data variable is the array **save_pal**. This array is used to save the current palette at the start of the program.

Under the application-specific routines in function **main( )**, the workstation is cleared using **v_clrwk( )**. Next function **check_table( )** is called. This function checks to see if a *color table* is supported by the VDI for this device. A color table is a VDI concept that provides a device-independent method for determining the color settings. There are two different levels of color manipulation in the ST. One is the VDI and the other is direct changes to the color palette. The VDI handles converting a color requested by the program to the best-matched color on the device. For example, on the Atari color display, the VDI uses a table to convert the VDI colors to the palette colors used by the display hardware. However, on a color slide producer, the VDI uses the lookup table to find the color set by the user and outputs this color to the device (slide producers generally have an extremely large range of colors). The other level of color graphics manipulation is through the Atari display hardware. This is done by actually accessing the color palette itself. By changing the color palette, you can alter the palette as a whole or set an individual color.

The **check_table( )** routine uses the **vq_extnd( )** function to do an extended inquiry. Note that the second parameter is a 1 for the **vq_extnd( )** function. This requests the extended values to be returned rather than simply the open workstation values. Compare this usage to the way the **vq_extnd( )** function is used in **set_screen_attr( )**. The returned values of the extended inquiry are put into the array elements. Element 5 contains the flag as to whether or not the color table is supported. Element 5 contains TRUE if a color table exists, FALSE otherwise. If the table is not supported, the VDI function **set_color( )** has no effect.

After checking the color table support, **main( )** continues by setting

the interior-fill mode to a solid fill using **vsf_interior( )**. Then function **show_palette( )** is used to draw a set of 16 rectangles with each rectangle drawn in a different VDI color.

The next function called by **main( )** is **change_palette( )**. This uses the VDI set color function, **vs_color( )**, to change a particular color of the palette. The **vs_color( )** function has three parameters. The first parameter is, as usual, the handle of the workstation. The second parameter is the color number to be changed. The last parameter is an array of three elements. The first element in the array is the red intensity, the second is the green intensity, and the third is the blue intensity.

The VDI color table is somewhat different than the Atari color palette. The VDI color table is as large as necessary for the VDI to support all of the colors available for that device. Each color has a red, green, and blue intensity; however, the intensity levels range from 0 to 1000 with 0 the lowest and 1000 the highest intensity. The intensity levels are relative to the actual levels that can be produced. For example, on the Atari color display, a VDI intensity of 1000 equates to an intensity level of 7 in the color palette. A palette intensity level of 5 is equivalent to a VDI intensity of 714. Thus, the VDI can support a device that has coarse intensity settings such as the color display, as well as a device with very fine intensity settings such as a color slide producer.

In **change_palette( )**, the first step is to inquire about color table entry number 3 using the **vq_color( )** function. The parameters of **vq_color( )** are the workstation handle, the color number, and a number which tells the function what values to return. A parameter of 0 (as used here) indicates that the set values of the color are to be returned. If the parameter is a 1, the "realized" values are returned. These are the values that are seen on the device, since the device probably does not have 1000 levels of intensities. The realized value shows what intensity value is actually shown on the screen. The last parameter of the function is a three-element array, which holds the intensity levels for each of the three colors.

The next step is to initialize the intensity levels for the red, green, and blue colors. After this initialization, **change_palette( )** loops through a series of color changes. These changes correspond to traveling around the color cube shown in Figure 6-7. Starting at the lower left corner, the first loop in **change_palette( )** increases the blue intensity, which is equivalent to moving up the left edge of the cube. The next loop moves diagonally across the top by increasing the red and green intensity levels. The third loop moves diagonally across the right face by decreasing the blue and green intensity levels. The final loop decreases the red intensity level to return to the starting position. Figure 6-7 shows eight intensity levels because these are the

realized intensity levels. When this program runs, a black rectangle goes through these color transformations. The last statement in **change_palette( )** restores color 3 to its original setting.

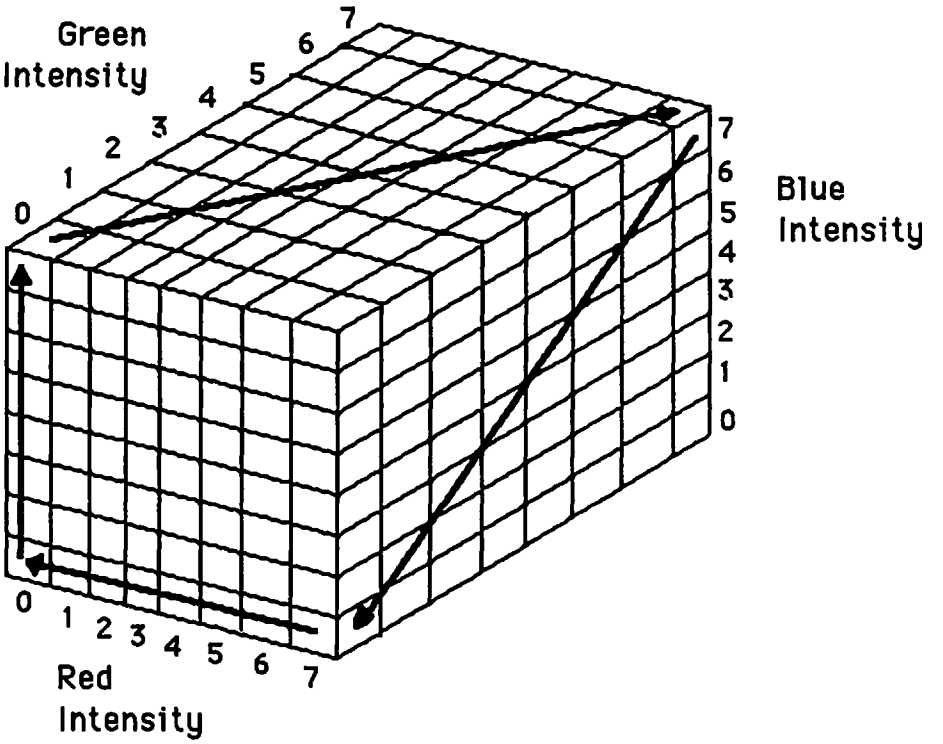


Figure 6-7 Color Cube

The next function called from **main( )** is the **rot_palette( )** function. This rotates colors within the palette by using XBIOS routines. The first thing to do is to save the current palette settings by calling the **Setcolor( )** function for each of the 16 color entries. The first parameter of **Setcolor( )** is the color index. The second is the intensity combination to place at that color entry. **Setcolor( )** returns the intensity value of the palette entry before it changes anything. If a negative value is used for the color intensity, no change is made. The negative parameter is used to read the current values and store them in the **save_pal** array. The next step is to rotate the palette colors. This is done through the **Setpalette( )** function. Function **Setpalette( )** has only one parameter: the address of a 16-word block of memory. **Setpalette( )** actually sets the 16 palette entries by placing those 16 words into the palette. The palette for the ST is actually a reserved portion of memory used by the display hardware.

To rotate the palette, a loop is used. The loop calls **Setpalette( )** with the address of the next element in array **pal_wheel**. Then function **show_palette( )** is called to draw the set of 16 squares. Actually, the **Setpalette( )** function affects the current display so the **show_palette( )** call is unnecessary. The reason the new palette colors take effect immediately is that the graphics hardware needs to access the palette to refresh the screen. Therefore, if you change the palette colors, the displayed colors also change on the next refresh cycle.

Now it's obvious why the **pal_wheel** array has 32 entries. **Set_palette( )** is called with the address of each of the first 16 elements in array **pal_wheel**. This address is used as the "base address" of a 16-word block in memory. Thus, for each address, there must be at least 16 entries defined following that address. The first call to **Set_palette( )** assigns the first through sixteenth elements in the array to the palette; the second call assigns the second through seventeenth elements in the array; and so on until the sixteenth call assigns the sixteenth through thirty-second elements in the array. At the end of **rot_palette( )**. **Set_palette( )** is called once more to restore the original palette colors.

When the program is running, notice that **rot_palette( )** does not display the colors in the order they are listed in the array. From the program listing where the **pal_wheel** array is defined, you would think: the first line of boxes displayed would be black, blue, green, and red; the second line cyan, magenta, yellow, and white; the next two lines various shades of gray. Instead of this predicted pattern, you see a mix of the various colors that are supposed to be shown. The reason is that program COLOR mixes the use of VDI output and XBIOS changes. The VDI is used in **show_palette( )** to draw the display and XBIOS is used to set the colors of the display. The VDI color indices are not the same as the palette color indices. The matching of the VDI and palette color indices is shown in Figure 6-8.

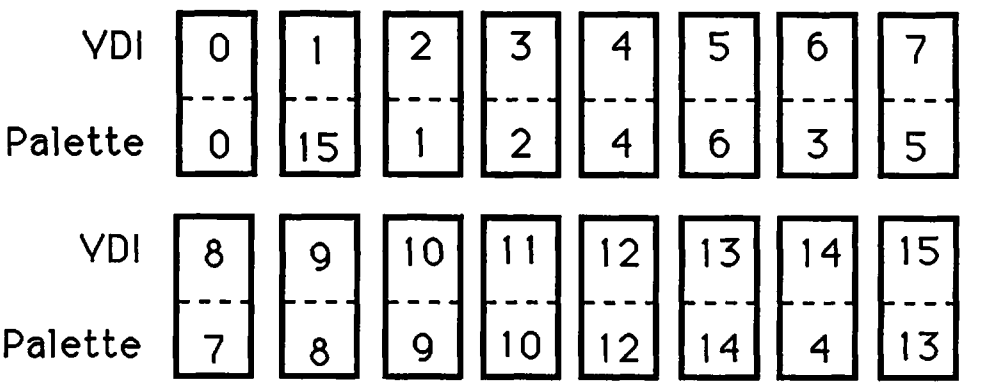


Figure 6-8 Map of VDI Color Index to Hardware Palette Index

The VDI indexes numbered 0 to 15 reference the actual palette numbers shown in the lower half of the boxes. Note that the background color (always color 0) does change in accordance with the sequence of colors in the **pal_wheel** array. This is to be expected because VDI color 0 and palette color 0 use the same color. Remember that color 0 is always used as the background color because objects are most easily erased by redrawing them with the background color. This is seen on the palette display where the box drawn in color 0 is not visible on the screen.

An interesting exercise is to take out the **show_palette( )** call in **rot_palette( )**. You see that the **Setpalette( )** call replaces the palette in between refresh cycles. You can also try adding a function that allows the user to enter a color index and a value, so that the user can manually change the color shown on the screen. See how the values entered affect the display. Remember to restore the original color palette; otherwise, you may not be able to read the desktop!

Program BOXES

The program BOXES produces colorful kinetic box art somewhat like program LINES1. It moves a box around the screen and the box appears to bounce off the edges. The size of the box cyclically grows and shrinks. As each new box is drawn, the palette is rotated to give an interesting psychedelic effect.

The program structure should look familiar (see Listing 6-2). Under the application-specific data, variables are defined to contain the lowest and highest x and y values of the drawing area. There are two 16-element arrays to hold the two palettes: the rotated palette and the original palette. The variable **max_color** holds the maximum number of palette entries. If you run the program in the medium resolution mode, you must set this number to 4.

In the application functions, the function **Rnd_rng( )** is the random number generator from an earlier program. The **change_color( )** function rotates the colors within the palette. The **draw_boxes( )** routine is the primary function in the program. To make the box appear to move in a straight line while it changes size, the center of the box is used as the reference point from which it is drawn. From the center of each box the coordinates of its opposite corners are calculated. The array **box** holds the corners of the box, **bcx** and **bcy** are the center x and y coordinates, **bsize** is the current size of the box (in pixels), **bdx** and **bdy** describe how the center coordinates of the box are going to move, and **bdsize** determines how the size of the box will change. The **cur_color** variable is the current drawing color.

Listing 6-2 Program BOXES

```
/*********************************************************************
     BOXES.C     Draw kinetic box art

     This program is similar to LINES1.C. This program draws
     boxes in color and changes the color palette to create
     visual effects.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

/* Constant values for drawing area */
int  x_lower,                      /* lowest x value */
     y_lower,                      /* lowest y value */
     x_upper,                      /* highest x value */
     y_upper;                      /* highest y value */

WORD pal_save[16],                 /* current palette */
     pal_wheel[16];                /* rotating palette */

WORD max_color = 16;               /* max colors displayable */
```

Listing 6-2 (continued)

```
/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;              /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)
          work_in[i] = 1;
     work_in[10] = 2;
     new_handle = phys_handle;     /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     v_clrwk(new_handle);          /* clear workstation */
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
     vq_extnd(screen_vhandle, 1, work_out);  /* get more info */
/*   max_color = work_out[4] * work_out[4];*/     /* planes ^ 2 */
     if (max_color < 2)            /* must have 2 colors */
          max_color = 2;
}

/*************************************************
     Application Functions
*************************************************/

long Rnd_rng(low, hi)
long low, hi;
```

Listing 6-2 (continued)

```
/*************************************************
Function: Generate a random number between low and hi, inclusive.
Input:    low  = lowest value in range.
          hi   = highest value in range.
Output:   Returns random number.
*************************************************/
{
     hi++;                          /* include hi value in range */
     return( (Random() % (hi - low)) + low);
}


change_color()
/*************************************************
Function: Rotates color palette.
Input:    None. Uses pal_wheel[].
Output:   None.
Notes:    Color index 0 is not changed. Generally you do not
          change this index because it is used as the
          background color.
*************************************************/
{
register        i;
register   WORD temp;


     temp = pal_wheel[1];                /* save first entry */
     for(i = 2; i < max_color; i++)      /* shift all entries down */
          pal_wheel[i-1] = pal_wheel[i];
     pal_wheel[max_color-1] = temp;      /* put first into last */
     Setpalette(pal_wheel);              /* change palette */
     return;
}


draw_boxes()
/*************************************************
Function: Do kinetic box art
Input:    None.
Output:   None.
*************************************************/
{

#define   BOX_MAX   40
#define   BOX_MIN   4

WORD box[4],                        /* array for box corners */
     bcx, bcy,                      /* center coords */
     bsize,                         /* current box size */
     bdx, bdy,                      /* deltas for box */
     bdsize;                        /* delta for size */

WORD cur_color;                     /* current drawing color */
```

Listing 6-2 (continued)

```
do                                  /* begin screen control loop */
{
     v_clrwk(screen_vhandle);       /* clear screen */

/* Initialize line corners */
     bsize     = BOX_MIN;
     bdsize    = 4;
     bcx  = Rnd_rng( (long)(x_lower+bsize), (long)(x_upper-bsize));
     bcy  = Rnd_rng( (long)(y_lower+bsize), (long)(y_upper-bsize));
     bdx       = Rnd_rng( -10L, 10L);
     bdy       = Rnd_rng( -10L, 10L);

     cur_color = 1;                 /* start color index */

/* Box drawing loop begins here */
     do
     {
          box[0] = bcx - bsize;     /* set corner coords */
          box[1] = bcy - bsize;
          box[2] = bcx + bsize;
          box[3] = bcy + bsize;
                                    /* set color to draw */
          vsf_color(screen_vhandle, cur_color);
          cur_color++;              /* set new color */
          if (cur_color >= max_color)
               cwr_color = 1;
          vr_recfl(screen_vhandle, box);      /* Draw box */

     /* Calculate new size */
          bsize += bdsize;

     /* check ranges */
          if (bsize < BOX_MIN)
          {
               bsize = BOX_MIN;
               bdsize = -bdsize;
          }
          if (bsize > BOX_MAX)
          {
               bsize = BOX_MAX;
               bdsize = -bdsize;
          }

     /* calculate new corners */
          bcx += bdx;
          bcy += bdy;

          if ((bcx - bsize) <= x_lower)
          {
               bcx = x_lower + bsize;
               bdx = -bdx;
```

Listing 6-2 (continued)

```
        }
        if ((bcx + bsize) >= x_upper)
        {
             bcx = x_upper - bsize;
             bdx = -bdx;
        }

        if ((bcy - bsize) <= y_lower)
        {
             bcy = y_lower + bsize;
             bdy = -bdy;
        }
        if ((bcy + bsize) >= y_upper)
        {
             bcy = y_upper - bsize;
             bdy = -bdy;
        }

        /* change colors */
        change_color();
    } while ( !Cconis() );              /* check if key pressed */
} while ( (Crawcin() & 0x7F) != 27);    /* escape key exits */
    return;
}

/***************************************************
    Main Program
***************************************************/

main()
{
int  ap_id;                         /* application init verify */
int  i;

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/***************************************************
    Initialize GEM Access
***************************************************/

    ap_id = appl_init();            /* Initialize AES routines */
    if (ap_id < 0)                  /* no calls can be made to AES */
    {                               /* use GEMDOS */
        Cconws('***> Initialization Error. <***\n');
        Cconws('Press any key to continue.\n');
        Crawcin();
        exit(-1);                   /* set exit value to show error */
    }
```

Listing 6-2 (continued)

```
     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */

/*************************************************
     Application Specific Routines
*************************************************/

                                   /* replace used for multicolors */
     vswr_mode(screen_vhandle, MD_REPLRCE);

/* set boundaries */
     x_lower = 10;
     y_lower = 10;
     x_upper = x_max - 10;
     y_upper = y_max - 10;

/* load palettes */
     for (i = 0; i < 16; i++)
     {
          pal_save[i] = Setcolor(i, -1);
          pal_wheel[i] = pal_save[i];
     }

/* start program */
     draw_boxes();                 /* Do box art */
     Setpalette(pal_save);         /* restore palette */

/*************************************************
     Program Clean-up and Exit
*************************************************/

     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
```

The flow in program BOXES is the same as the flow for program LINES. An outer loop initializes the screen and starting variables. The inner loop continually produces the kinetic image until a key is pressed. The keypress causes the inner loop to exit. The outer loop checks if the key was the ESC key. If not, the outer loop repeats and a new kinetic image is produced. If the ESC key was pressed, the function **draw_boxes( )** exits back to **main( )** and the program ends.

In **draw_boxes( )**, the outer loop starts by clearing the screen and initializing the first box. The variable **bsize** is set to the minimum box size: 4 pixels as defined in the program. This measurement represents the distance from the center to the edge of the box so that

the box is actually 8 pixels wide and 8 pixels tall. The change in box size, **bdsize**, is set to 4. The center x and y coordinates and the changes in those coordinates are randomly selected. The current color is set at 1. The inner loop now begins.

The first step inside this loop is to set the coordinates of the box corners. The color is set to the current drawing color and then incremented so that the next time through the loop a different color is used. After incrementing the color index, a test is made to ensure that the color index is always less than or equal to the maximum number of colors available. Once the box is drawn, the new box size is determined and checked to be sure it does not exceed the size limits. The new box center is calculated and checked to see if the new box fits within the limits of the screen. Next the colors displayed on the screen are changed by calling **change_color( )**. The loop ends by checking if any keys have been pressed.

Looking at the application-specific functions of **main( )**, the first instructions are to set the writing mode to replace and set the boundaries of the screen. Note that the writing mode XOR is not used as it was in LINES. When using color output, the writing modes have a slightly different effect than in monochrome output because the XOR mode has a strange effect on the colors: they all come out the same! The replace mode simply draws over whatever is on the screen. The transparent mode changes all colors except the background color to whatever color is currently set. The reverse transparent mode changes only the background color to the current color.

After the mode is set, the palettes are loaded and **draw_boxes( )** is called. When **draw_boxes( )** returns, the original palette is restored and the program exits.

Try using the different writing modes in program BOXES.

CHAPTER SEVEN

Moving Targets

This chapter covers the VDI concept called a *raster* and how and where it is used. Once you are familiarized with the raster, you can use it and many other VDI features to create an example animation program.

The Raster

What is a raster? A raster is a more generalized form of the bit map. It is a "rectangular" block of memory used to represent a graphic image. A raster, being rectangular, has a width and a height. The width is measured in two ways: in pixels that correspond to individual bits in memory and in words. The width must be a whole integral of the word size. For example, a raster may be 1, 2, 3, or more words in width. This means that the width must be 16, 32, 48, or more pixels in size. The width in pixels must be a multiple of 16, which is an integral word size. The height is measured strictly in rows of pixels. For color rasters, a third measurement, the number of planes is also used. Look at Figure 7-1 to see how the raster measurements are made.

The raster is similar to a screen bit map. The first word in memory contains the first 16 bits in the first row from the left side. The words continue across and down the raster just as in the bit map. The raster also has a coordinate system like the bit map. The upper left corner is considered the origin with a coordinate pair (0,0). The x coor-

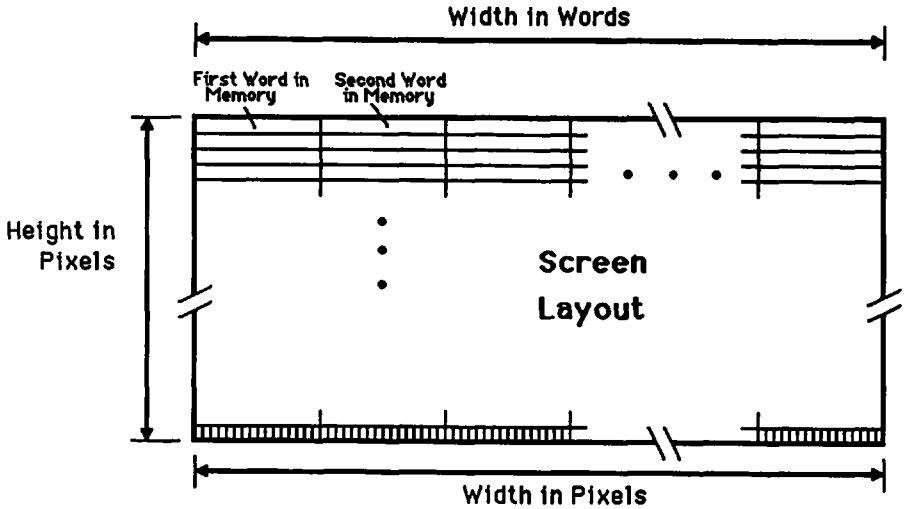


Figure 7-1 Raster Measurements

dinates increase as you move from left to right, and the y coordinates increase as you move from top to bottom.

Primarily, the raster is a bit map which can have a user-specified size. You may wonder why a raster is needed. A raster can provide a rapid means for transferring a graphics image from one place in memory to another. For example, if you want to draw the floppy disk icon on the screen, you can simply define the icon in a raster and copy that image to any location on the screen. This is much faster than redrawing the rectangles, lines, and letters to the screen each time you want to display a floppy disk icon; redrawing could take several seconds. Another way the raster is used is to store a portion of the screen in memory, use that area of the screen, and then replace the original image. This is the general procedure used to display a menu list. When the menu drops down, the area under the menu is saved in a raster-type portion of memory. When the user has finished the menu selection, the old screen image is copied to the screen over the area covered by the menu. Thus, the screen is restored to its original state.

Using a Raster

You need two things to use a raster: a portion of memory to be used for the raster contents and a *Memory Form Definition Block* (MFDB). This MFDB describes the various attributes associated with the raster. These attributes are the width and height of the raster, a

pointer to the raster location in memory, the number of planes used by the raster, and a flag telling the program what format the raster uses.

The Memory Form Definition Block

For this book's programming purposes, the MFDB is defined as a structure. In the listing of the program RASTER under the GEM application overhead, the type definition for structure **mfdbstr** contains the necessary elements. There is also a structure type definition in header file gemdefs.h called **FDB**. This is exactly the same structure as MFDB in RASTER, but the member names of FDB are abbreviated and not as easy to understand.

```
typedef struct mfdbstr
{
    char *addr;          /* address of the raster area */
    WORD wide;           /* width of the raster in pixels */
    WORD high;           /* height of the raster in pixels */
    WORD word_width;     /* width of the raster in words */
    WORD format;         /* standard or device-specific */
    WORD planes;         /* number of planes in raster */
    WORD reserv1,        /* reserved for future use */
         rsserv2,
         reserv3;
} MFDB;
```

The address is a pointer to the location in memory where the raster resides. Element *wide* is the width of the raster in *pixels*; element *high* is the height of the raster in rows (that is, pixels); and element **word_width** is the width of the raster in words. (Remember that the raster must have a width that is an even multiple of one word.) The element *format* determines the layout of the raster. If format is 0, the layout of the raster is in device-specified format. If format is 1, the layout is in standard format. The element *planes* determines the number of planes held in the raster.

Raster Formats

A raster format determines how the planes of the raster are stored in memory. The VDI uses two methods for storing planes. In the first method, which you have already seen, the first word of plane 0 is followed by the first word of plane 1 and so on. This is known as an

Listing 7-1 Program RASTER

```
/*********************************************************************
     RASTER.C Raster exercising program

     This program demonstrates raster operations. Use this program
     to experiment with the raster operations.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptscut[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

typedef   struct    mfdbstr
{
     char      *addr;              /* address of raster area */
     WORD      wide;               /* width of raster in pixels */
     WORD      high;               /* height of raster in pixels */
     WORD      word_width;         /* width of raster in words */
     WORD      format;             /* standard or device specific */
     WORD      planes;             /* number of planes in raster */
                                   /* reserved for future use */
     WORD      reserv1, reserv2, reserv3;
} MFDB;

/*************************************************
     Application-Specific Data
*************************************************/
```

Listing 7-1 (continued)

```
MFDB chexMFDB,
     stripeMFDB,
     blockMFDB,
     scrMFDB,
     tempMFDB;

WORD block[16] = {  0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                    0xFFFF,
                }:

WORD stripe[16] = { 0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                    0xAAAA,
                }:

WORD chex[32] = {   0xCCCC,         /* plane 0 */
                    0xCCCC,
                    0x3333,
                    0x3333,
                    0xCCCC,
                    0xCCCC,
                    0x3333,
                    0x3333,
                    0xCCCC,
                    0xCCCC,
```

Listing 7-1 (continued)

```
                    0x3333,
                    0x3333,
                    0xCCCC,
                    0xCCCC,
                    0x3333,
                    0x3333
               };                    /* plane 1 will be set to 0s */

WORD temp[16][4];                    /* allow for 4 planes */

/*************************************************
     GEM-related Functions
*************************************************/


WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)
          work_in[i] = 1;
     work_in[10] = 2;
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}
```

Listing 7-1 (continued)

```
/*************************************************
     Application Functions
*************************************************/

raster_form()
/*************************************************
Function: Demonstrates different raster formats.
Input:    None. Uses global MFDBs and rasters listed above.
Output:   Sets fields in MFDBs which should be reset before
          doing anything else.
*************************************************/
{
int  i;

/* Checkerboard raster */
     chexMFDB.addr       =    (char *)chex;
     chexMFDB.wide       =    16;
     chexMFDB.high       =    16;
     chexMFDB.word_width =    1;
     chexMFDB.format     =    1;        /* standard format */
     chexMFDB.planes     =    2;

/* Temporary raster area */
     tempMFDB.addr       =    (char *)temp;
     tempMFDB.wide       =    16;
     tempMFDB.high       =    16;
     tempMFDB.word_width =    1;
     tempMFDB.format     =    0;        /* will be set by vr_trnfm() */
     tempMFDB.planes     =    2;        /* mwst be same as source */

     vr_trnfm(screen_vhandle, &chexMFDB, &tempMFDB);
     printf("Transform from Standard to Device-specific\n");
     for (i = 0; i < 32; i++)
          printf("%2d: Standard %6x  Device %6x\n",
               i, chex[i], temp[i]);
     Crawcin();

     chexMFDB.format = 0;          /* set to device-specific */
     vr_trnfm(screen_vhandle, &chexMFDB, &tempMFDB);
     printf("Transform form Device-specific to Standard\n");
     for (i = 0; i < 32; i++)
          printf("%2d: Device %6x  Standard %6x\n",
               i, chex[i], temp[i]);
     Crawcin();
}

set_up_rasters()
/*************************************************
Function: Sets wp global MFDBs to point to rasters.
Input:    None. Uses global MFDBs and rasters listed above.
Output:   Sets appropriate fields in MFDBs.
*************************************************/
{
```

Listing 7-1 (continued)

```
/* Checkerboard raster */
     chexMFDB.addr        =    (char *)chex;
     chexMFDB.wide        =    16;
     chexMFDB.high        =    16;
     chexMFDB.word_width  =    1;
     chexMFDB.format      =    0;        /* must be device-specific */
     chexMFDB.planes      =    1;        /* monochrome has 1 plane */

/* Stripe raster */
     stripeMFDB.addr          =    (char *)stripe;
     stripeMFDB.wide          =    16;
     stripeMFDB.high          =    16;
     stripeMFDB.word_width    =    1;
     stripeMFDB.format        =    0;
     stripeMFDB.planes        =    1;   /* monochrome has 1 plane */

/* Block raster */
     blockMFDB.addr           =    (char *)block;
     blockMFDB.wide           =    16;
     blockMFDB.high           =    16;
     blockMFDB.word_width     =    1;
     blockMFDB.format         =    0;
     blockMFDB.planes         =    1;   /* monochrome has 1 plane */

/* Screen raster area */
     scrMFDB.addr         =    (char *)Logbase();
     scrMFDB.wide         =    640;
     scrMFDB.high         =    400;
     scrMFDB.word_width   =    40;
     scrMFDB.format       =    0;
     scrMFDB.planes       =    1;        /* monochrome has 1 plane */
     if (screen_rez == 0)                /* low resolution */
     {
          scrMFDB.wide    =    320;
          scrMFDB.high    =    200;
          scrMFDB.word_width = 20;
          scrMFDB.planes =    4;
     }
     else if (screen_rez == 1)           /* medium resolution */
     {
          scrMFDB.high    =    200;
          scrMFDB.planes =    2;
     }

/* Temporary raster area */
     tempMFDB.addr        =    (char *)temp;
     tempMFDB.wide        =    16;
     tempMFDB.high        =    16;
     tempMFDB.word_width  =    1;
     tempMFDB.format      =    0;
     tempMFDB.planes      =    scrMFDB.planes;    /* same as screen */
}
```

Listing 7-1 (continued)

```
raster_test()
/*************************************************
Function: Draw patterns using rasters.
Input:    None. Uses MFDBs and rasters defined above.
Output:   None. No changes to MFDBs and rasters except temp.
*************************************************/
{
WORD pxy[8],
     color_index[2];
int  pass;                          /* use to loop twice */

     color_index[0] = 1;            /* Foreground color value */
     color_index[1] = 0;            /* Background color value */


for  (pass = 0; pass < 2; pass++)   /* first pass uses vrt_cpyfm */
{                                   /* second pass uses vro_cpyfm */
                                    /* set background pattern */
     v_clrwk(screen_vhandle);
     vsf_interior(screen_vhandle, 2);
     vsf_style(screen_vhandle, 2);
     pxy[0] = pxy[1] = 10;
     pxy[2] = pxy[3] = 150;
     vr_recfl(screen_vhandle, pxy);


/* Save portion of screen in temp */
     pxy[0] = 100;                  /* source x1 coord */
     pxy[1] = 100;                  /* source y1 coord */
     pxy[2] = 115;                  /* source x2 coord */
     pxy[3] = 115;                  /* source y2 coord */
     pxy[4] = 0;                    /* dest x1 coord */
     pxy[5] = 0;                    /* dest y1 coord */
     pxy[6] = 15;                   /* dest x2 coord */
     pxy[7] = 15;                   /* dest y2 coord */


     vro_cpyfm(screen_vhandle, S_ONLY, pxy,
          &scrMFDB, &tempMFDB);


/* Draw checkerboard pattern in saved area */
     pxy[0] = pxy[1] = 0;
     pxy[2] = pxy[3] = 15;
     pxy[4] = pxy[5] = 100;
     pxy[6] = pxy[7] = 115;
     if (pass)
          vro_cpyfm(screen_vhandle, S_ONLY, pxy,
               &chexMFDB, &scrMFDB);
     else
          vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
               &chexMFDB, &scrMFDB, color_index);
     Crawcin();
```

Listing 7-1 (continued)

```
/* Draw stripe pattern over part of checkerboard */
     pxy[0] = pxy[1] = 0;
     pxy[2] = pxy[3] = 15;
     pxy[4] = pxy[5] = 110;
     pxy[6] = pxy[7] = 125;
     if (pass)
          vro_cpyfm(screen_vhandle, S_ONLY, pxy,
               &stripeMFDB, &scrMFDB);
     else
          vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
               &stripeMFDB, &scrMFDB, color_index);
     Crawcin();

/* Draw block pattern */
     pxy[0] = pxy[1] = 0;
     pxy[2] = pxy[3] = 15;
     pxy[4] = pxy[5] = 132;
     pxy[6] = pxy[7] = 147;
     if (pass)
          vro_cpyfm(screen_vhandle, S_ONLY, pxy,
               &blockMFDB, &scrMFDB);
     else
          vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
               &blockMFDB, &scrMFDB, color_index);
     Crawcin();

/* Demonstrate different rectangle sizes with block pattern */
     pxy[0] = pxy[1] = 0;          /* destination wider than source */
     pxy[2] = pxy[3] = 10;
     pxy[4] = pxy[5] = 10;
     pxy[6] = 50;
     pxy[7] = 20;
     if (pass)
          vro_cpyfm(screen_vhandle, S_ONLY, pxy,
               &blockMFDB, &scrMFDB);
     else
          vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
               &blockMFDB, &scrMFDB, color_index);
     Crawcin();

     pxy[0] = pxy[1] = 0;          /* destination longer than source */
     pxy[2] = pxy[3] = 10;
     pxy[4] = 60;
     pxy[5] = 10;
     pxy[6] = 70;
     pxy[7] = 50;
     if (pass)
          vro_cpyfm(screen_vhandle, S_ONLY, pxy,
               &blockMFDB, &scrMFDB);
     else
          vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
               &blockMFDB, &scrMFDB, color_index);
     Crawcin();
```

Listing 7-1 (continued)

```
/* Replace saved portion of screen from temp */
     pxy[0] = pxy[1] = 0;
     pxy[2] = pxy[3] = 15;
     pxy[4] = pxy[5] = 100;
     pxy[6] = pxy[7] = 115;
     vro_cpyfm(screen_vhandle, S_ONLY, pxy,
          &tempMFDB, &scrMFDB);
     Crawcin();
} /* end pass loop */
}


color_test()
/*************************************************
Function: Show use of raster copy and writing modes with color.
Input:    None. Uses chex[], chexMFDB, and scrMFDB.
Output:   None.
*************************************************/
{
WORD      pxy[8],
          color_index[2];


     v_clrwk(screen_vhandle);            /* clear screen */
     pxy[0] = 10;   pxy[1] = 10;
     pxy[2] = 300;  pxy[3] = 150;
     vsf_interior(screen_vhandle, 1);    /* solid fill */
     vsf_color(screen_vhandle, 4);       /* use color 4 to fill */
     vr_recfl(screen_vhandle, pxy);      /* draw rectangle */

/* use replace mode */
     pxy[0] = 0;    pxy[1] = 0;
     pxy[2] = 15;   pxy[3] = 15;
     pxy[4] = 50;   pxy[5] = 20;
     pxy[6] = 65;   pxy[7] = 35;
     if (color_screen)
     {
          color_index[0] = 2;      /* color for 1 bits */
          color_index[1] = 3;      /* color for 0 bits */
     }
     else
     {
          color_index[0] = 1;
          color_index[1] = 0;
     }
     vrt_cpyfm(screen_vhandle, MD_REPLACE, pxy,
          &chexMFDB, &scrMFDB, color_index);


/* use transparent mode */
     pxy[4] = 100;  pxy[5] = 20;
     pxy[6] = 115;  pxy[7] = 35;
     vrt_cpyfm(screen_vhandle, MD_TRANS, pxy,
          &chexMFDB, &scrMFDB, color_index);
```

Listing 7-1 (continued)

```
/* use XOR mode */
     pxy[4] = 150;  pxy[5] = 20;
     pxy[6] = 165;  pxy[7] = 35;
     vrt_cpyfm(screen_vhandle, MD_XOR, pxy,
          &chexMFDB, &scrMFDB, color_index);

/* use reverse transparent mode */
     pxy[4] = 200;  pxy[5] = 20;
     pxy[6] = 215;  pxy[7] = 35;
     vrt_cpyfm(screen_vhandle, MD_ERASE, pxy,
          &chexMFDB, &scrMFDB, color_index);

     return;
}

/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }
     screen_phandle =               /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Get screen attributes */

/*************************************************
     Application Specific Routines
*************************************************/

     v_clrwk(screen_vhandle);       /* clear workstation */
     raster_form();
     set_up_rasters();
```

Listing 7-1 (continued)

```
     raster_test();
     color_test();

/**************************************************
     Program Clean-up and Exit
**************************************************/

/* Wait for keyboard before exiting program */
     Crawcin();                    /* GEMDOS character input */
     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
/**************************************************/
```

interleaved format, which the VDI has termed the device-specific format (see Figure 7-2). The second format has all the words for plane 0 set consecutively in memory. This is followed by all the words for plane 1, all the words for plane 2, and so on until all planes have been listed. This is a *contiguous* format and is called the standard format in the VDI (see Figure 7-3).

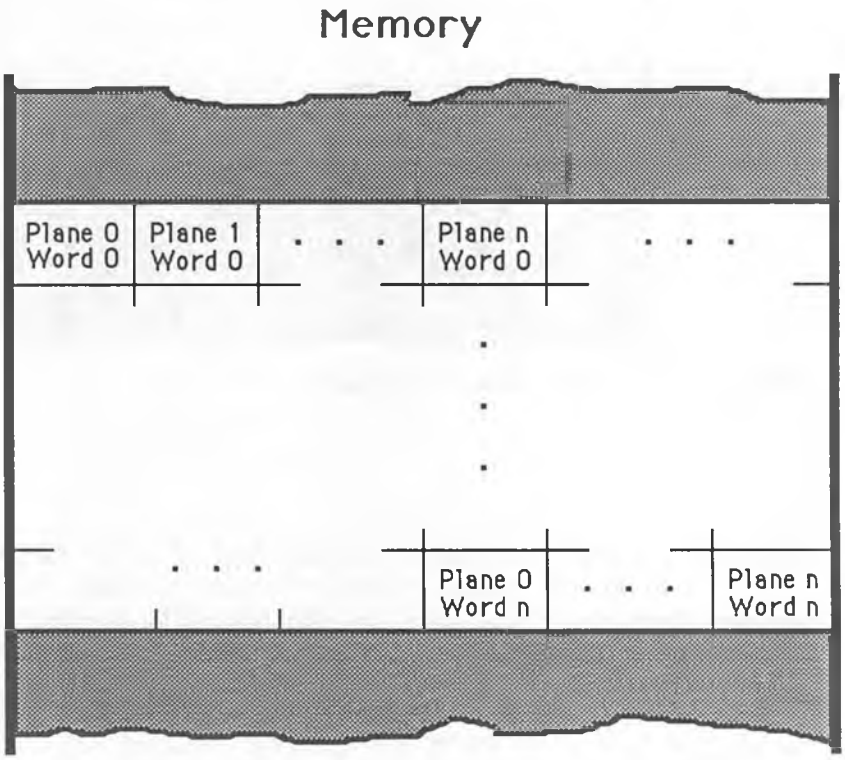


Figure 7-2 Device-Specific Format Plane Layout for Atari ST

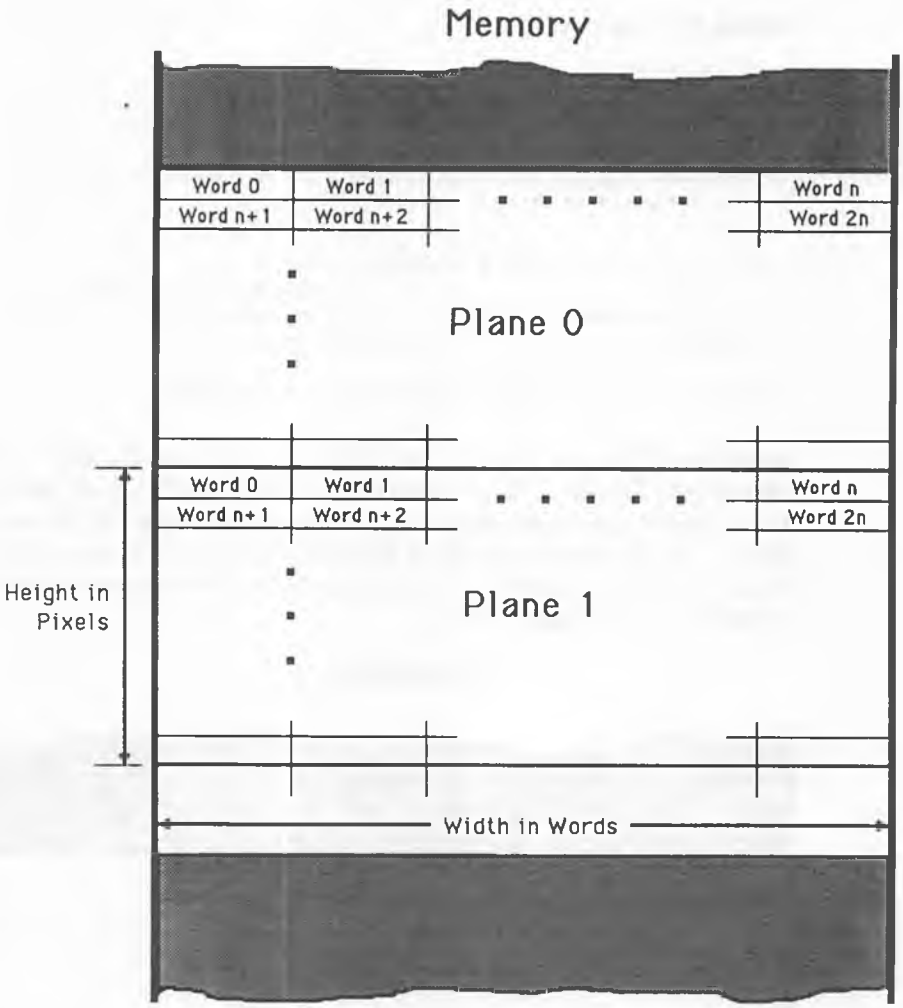


Figure 7-3 Standard Format Plane Layout

The device-specific format is said to be device-specific because this format may be different on another computer system. The format shown here is the one used on the Atari ST. Also note that a monochrome (monoplane) raster is the same in either device-specific or standard format. The program RASTER will demonstrate the difference between standard and device-specific formats and how to use a transformation function that can convert from standard to device-specific format and vice versa.

Color

The implementation of color in a raster is the same as in the screen bit map. Each plane contributes one bit toward an index in the VDI color table for that particular pixel. The default color indices are listed in Table 7-1.

Table 7-1: Pixel Value to Color Index Mapping

Pixel Value	*Color Index*
0000	0
0001	2
0010	3
0011	6
0100	4
0101	7
0110	5
0111	8
1000	9
1001	10
1010	11
1011	14
1100	12
1101	15
1110	13
1111	1

Using the Rasters in a Program

To use a raster in a program, you must set up the raster and its MFDB. You must also use one of the two raster copying functions to get the image to show on the screen. Essentially you are copying the raster from its original location to the screen raster (bit map). The first copying function is called an *opaque* raster copy. The second is a *transparent* raster copy. The opaque copy function transfers the image from a source raster to a destination raster pixel for pixel. Thus, it overwrites the destination raster with the source raster. The overwriting result is governed by the writing mode used for the copy function.

Opaque Copy Raster Function

The opaque copy raster function is called from a C program as follows:

vro_cpyfm(handle, wr_mode, pxyarray, psrcMFDB, pdesMFDB)

The parameters in the opaque copy function start with the workstation handle. This parameter does not have any effect on the raster copy but is included for consistency with all other VDI function calls. The second parameter is the writing mode, which determines how the raster is copied. The third parameter is an array describing the area to copy from and the area to copy to. The last two parameters of the function are the addresses of the source MFDB and the destination MFDB, respectively.

There are 16 different copy modes that perform a variety of logic operations between the source bit and destination bit. Table 7-2 shows the logic operation associated with each mode. In the table, "S" stands for the source bit and "D" stands for the destination bit. The result of the operation is placed in the destination raster. Several modes are worth special note. Mode 0 is the clear mode. It turns all bits in the destination raster to 0. Mode 3 is the replace mode, which exactly copies the contents of the source raster into the contents of the destination raster without regard to the contents of the destination raster. Mode 4 maps the destination bits to 1 if the source bit has a value of 0 and the destination bit has a value of 1; otherwise, the destination bit is set to 0. Mode 5 leaves the destination raster unchanged. Mode 10 inverts the existing destination raster. Mode 12 inverts the source raster. Mode 15 sets all bits in the destination raster to 1. Clearly, the writing mode is very significant. Since the opaque copy makes copies pixel by pixel, changing the writing mode can produce some rather interesting results, such as changing the color of the image (when working with planes) or changing the image itself.

Table 7-2: Opaque Raster Copy Logic Operations

Mode	*Operation*	*Mode*	*Operation*
0	set to 0	8	~(S | D)
1	S & D	9	~(S ^ D)
8	S & ~D	10	~D
3	S	11	S | ~D
4	~S & D	12	~S
5	D	13	~S | D
6	S ^ D	14	~ (S & D)
7	S | D	15	set to 1

The third parameter of **vro_cpyfm( )** is an array that holds the coordinates of two rectangles. The first four elements locate a rectangular area within the source raster. This denotes the area from which

the pixels are copied, and allows you to copy all or only part of a raster. The last four elements of the array outline a rectangular area in the destination raster where the source pixels are to be copied. With this option, you may locate your raster image anywhere within the destination raster. In the earlier example of copying a floppy disk icon to the screen, the destination raster (the screen) is obviously much bigger than the source raster. By specifying the coordinates of the rectangular area in the destination raster, you can map the icon anywhere on the screen.

Note that since the source and destination rasters may be the same (for example, copying from one portion of the screen to another), you may wind up in a situation where the source and destination rectangles overlap. This situation has been accounted for by the VDI so that you don't wind up changing a portion of the source raster before it has been copied. The VDI guarantees that any area in the source raster is not changed until it has been copied to the destination raster. In other words, the source raster is not "drawn over" until the corresponding area in the destination raster has already been copied.

A few other notes about the opaque copy function. The opaque raster copy function does not perform any transformations or rotations. Thus, if you have a source raster in device-specific format, the destination raster must also be in device-specific format. Also, all copying is done strictly *pixel by pixel*. Finally, if the source and destination rectangles are not the same size, the VDI manual states that the VDI uses the destination for a pointer and the source rectangle for the resulting size. In program RASTER, the results of this operation are seen. It is left to the reader to determine how well this option works. On the author's version of GEM, this feature was apparently not working properly.

Transparent Copy Raster Function

The other copy function, transparent copy raster, is used to create a color image from a monochrome raster. The function call from a C program is as follows:

vrt_cpyfm(handle, wr_mode, pxyarray,
psrcMFDB, pdesMFDB, color_index)

The parameters are similar to the opaque copy function. The first parameter is the workstation handle. The second parameter is the writing mode. The third parameter is the source and destination rectangle array. The next two parameters are pointers to the source and destination MFDBs, respectively. The last parameter is a two-element array that determines the color output.

The writing mode for the transparent copy is different from the

writing mode for the opaque copy function. The writing modes are the same used for standard VDI functions: replace, transparent; XOR, and reverse transparent. These modes are used in conjunction with the color index array to produce color output. Color index array element 0 is used to determine the color of the foreground, and element 1 is used to determine the background color. The value of the array element is the color index in the VDI color table.

In replace mode, all pixels with a value of 1 in the source raster yield a pixel with color **color_index[0]** in the destination, and all pixels with a 0 value have color **color_index[1]** at the destination. In transparent mode, a 1 bit in the source produces a pixel of **color_index[0]** in the destination. Source bits with a 0 value have no effect on the destination pixel. In XOR mode, an XOR operation is performed between each pixel of the source raster and each bit in *each* of the destination planes. The color index array is not used at all with this mode. Reverse transparent mode is the reverse of transparent mode; wherever the source raster has a pixel set to 0, **color_index[1]** is output to the destination. Source pixels set to 1 have no effect on the destination.

Raster Conversion

To convert a raster from device-specific format to standard format (or vice-versa), VDI provides a transform form function:

vr_trnfm(handle, psrcMFDB, pdesMFDB)

The parameters are the workstation handle, the address of the source MFDB, and the address of the destination MFDB. The values held in the source MFDB determine how the function operates. The source MFDB contains the number of planes to be transformed and the format to be used for the destination. If the source raster is in standard format, the destination is converted to device-specific format; otherwise, the destination is converted to standard format. The destination MFDB has its format field set to the resulting format by the transform function, but all other parameters of the destination MFDB must be manually set before the function call because they are not altered.

Program RASTER

Program RASTER demonstrates the use of rasters and raster copy functions. In the GEM-application overhead in Listing 7-1, there is the usual overhead plus the type definition for the MFDB. As men-

tioned earlier, our own structure is used because it is much easier to read and understand than the one included in the GEM header file gemdefs.h.

Under the application-specific data, there are five different MFDBs defined. This program demonstrates how to create and copy raster images and the effects of various copying and writing modes. There are three defined rasters: a checkerboard, a stripe, and a solid block. The other two MFDBs are used for the screen and for temporary storage of screen contents, respectively.

Four arrays are used to hold the raster images. The block and stripe rasters are 16-by-16 pixels. The checkerboard raster contains two planes of that size. Therefore, the array to hold the block and stripe patterns is 16 words long because the rasters are 16 pixels (1 word) wide by 16 pixels high. The array to hold the checkerboard raster is 32 words long (2 planes of 16 rows each). The temporary raster must be able to hold four planes so that it can store low resolution color images from the screen. The numbers used to initialize the arrays represent the raster images held by arrays. The process for converting from a pixel image to a hexadecimal value is shown in Figures 7-4 through 7-6.

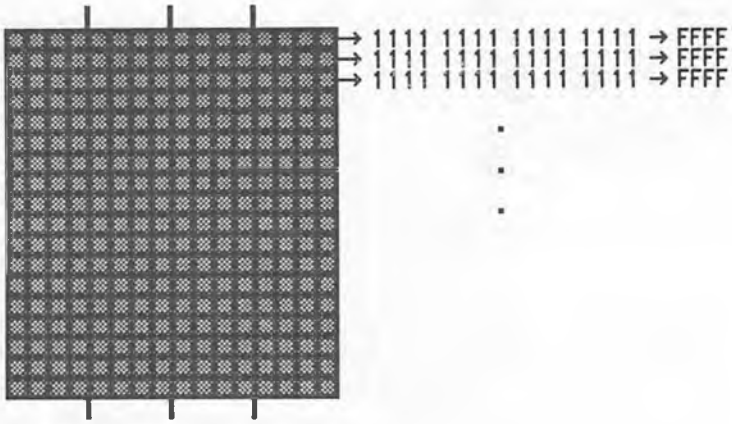


Figure 7-4 Block Pattern

In these figures, the three different rasters have been drawn in 16-by-16 grids. The block pattern consists of filling in every bit. The stripe pattern consists of vertical lines, and the checkerboard pattern is a checkerboard. Since C does not have binary constants, you need to represent the bit images in either decimal, octal, or hexadecimal numbers. The easiest conversion is to hexadecimal numbers. To do this, the "on" bits of the image are given a value of 1 and the "off" bits a value of 0. Each row in the image is divided into 4-bit groups, and each 4-bit group is converted into its hexadecimal code. This

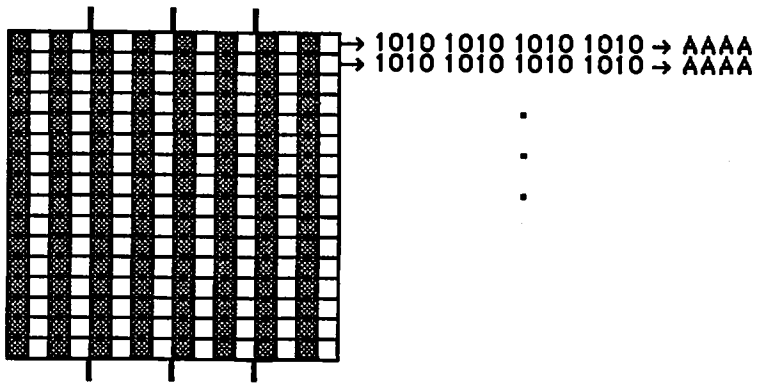


Figure 7-5 Stripe Pattern

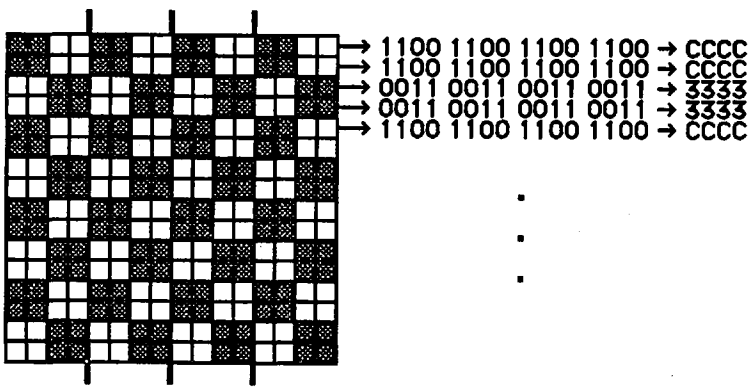


Figure 7-6 Checkerboard Pattern

transformation from pixels to bits to hexadecimal is shown in each of the figures. For each array element, then, simply fill in the appropriate hexadecimal value. Notice that the checkerboard array, which is supposed to hold 32 elements, has only 16 elements defined (representing one plane out of a possible two). The remaining elements are automatically set to 0 by the C compiler.

In **main( )**, the program performs the usual initialization and clears the workstation. Four application routines are then called. The first, **raster_form( )**, demonstrates the differences between the standard raster format and the device-specific format. The routine **set_up_rasters( )** initializes all the MFDBs used in the program. The third function, **raster_test( )**, provides a comparison between the opaque and transparent raster copy functions and an outline for the user to experiment with these copy functions. The final function, **color_test( )**, shows how the different writing modes affect the transparent copy raster function.

The routine **raster_form( )** uses the checkerboard raster to go between the standard format and the device-specific format. This function first sets up the MFDBs. The variable **chexMFDB** is set to point to the start of the **chex** array, which holds the checkerboard image raster. Its width and height are 16 pixels, the word width is 1, and the format is set to the standard format denoted by the value 1. For this test, the program uses the two planes of the checkerboard raster. The temporary raster MFDB is set up with the same attributes except that it points to the temporary array. The **vr_trnfm( )** function takes the raster pointed at by **chexMFDB** and places it into the raster pointed at by **tempMFDB**. The 32 elements (rows) of the **chex** and **temp** arrays are then printed out to show the difference between the standard and device-specific format of the two rasters. Note that the numbers in the **printf( )** statements are printed in hexadecimal notation so they can easily be compared to their initial settings at the top of the program.

Standard Format			Address	Device-specific Format		
Plane	Word	Value	(in bytes)	Plane	Word	Value
0	0	CCCC	Base	0	0	CCCC
0	1	CCCC	Base+2	1	0	0000
0	2	3333	Base+4	0	1	CCCC
0	3	3333	Base+6	1	1	0000
0	4	CCCC	Base+8	0	2	3333
0	5	CCCC	Base+10	1	2	0000
0	6	3333	Base+12	0	3	3333
0	7	3333	Base+14	1	3	0000
0	8	CCCC	Base+16	0	4	CCCC
0	9	CCCC	Base+18	1	4	0000
0	10	3333	Base+20	0	5	CCCC
0	11	3333	Base+22	1	5	0000
0	12	CCCC	Base+24	0	6	3333
0	13	CCCC	Base+26	1	6	0000
0	14	3333	Base+28	0	7	3333
0	15	3333	Base+30	1	7	0000
1	0	0000	Base+32	0	8	CCCC
1	1	0000	Base+34	1	8	0000
1	2	0000	Base+36	0	9	CCCC
1	3	0000	Base+38	1	9	0000
1	4	0000	Base+40	0	10	3333
1	5	0000	Base+42	1	10	0000
1	6	0000	Base+44	0	11	3333
1	7	0000	Base+46	1	11	0000
1	8	0000	Base+48	0	12	CCCC
1	9	0000	Base+50	1	12	0000
1	10	0000	Base+52	0	13	CCCC
1	11	0000	Base+54	1	13	0000
1	12	0000	Base+56	0	14	3333
1	13	0000	Base+58	1	14	0000
1	14	0000	Base+60	0	15	3333
1	15	0000	Base+62	1	15	0000

Figure 7-7 Transforming a Raster from Standard to Device-Specific Format

The transformation occurs as shown in Figure 7-7. The transformation involves taking the contiguous words in discrete planes (standard format) and converting them into interleaved planes (device-specific format). The **chex** array starts in standard format with all

words of plane 0 coming first, followed by all words for plane 1. The **temp** array holds the checkerboard pattern in device-specific format with word 0 of plane 0, followed by word 0 of plane 0, then word 1 of plane 0, and so on.

After printing the results of this first transformation, function **raster_form( )** waits for the user to press a key and then performs the opposite transformation. The format field of **chexMFDB** is set to indicate that the **chex** raster is in device-specific format. Although the actual data has not been changed, you are telling the VDI that the data is in device-specific format. The transformation function is called to convert the raster from device-specific format to standard format. The results are shown in Figure 7-8.

Device-specific Format			Address	Standard Format		
Plane	Word	Value	(in bytes)	Plane	Word	Value
0	0	CCCC	Base	0	0	CCCC
1	0	CCCC	Base+2	0	1	3333
0	1	3333	Base+4	0	2	CCCC
1	1	3333	Base+6	0	3	3333
0	2	CCCC	Base+8	0	4	CCCC
1	2	CCCC	Base+10	0	5	3333
0	3	3333	Base+12	0	6	CCCC
1	3	3333	Base+14	0	7	3333
0	4	CCCC	Base+16	0	8	0000
1	4	CCCC	Base+18	0	9	0000
0	5	3333	Base+20	0	10	0000
1	5	3333	Base+22	0	11	0000
0	6	CCCC	Base+24	0	12	0000
1	6	CCCC	Base+26	0	13	0000
0	7	3333	Base+28	0	14	0000
1	7	3333	Base+30	0	15	0000
0	8	0000	Base+32	1	0	CCCC
1	8	0000	Base+34	1	1	3333
0	9	0000	Base+36	1	2	CCCC
1	9	0000	Base+38	1	3	3333
0	10	0000	Base+40	1	4	CCCC
1	10	0000	Base+42	1	5	3333
0	11	0000	Base+44	1	6	CCCC
1	11	0000	Base+46	1	7	3333
0	12	0000	Base+48	1	8	0000
1	12	0000	Base+50	1	9	0000
0	13	0000	Base+52	1	10	0000
1	13	0000	Base+54	1	11	0000
0	14	0000	Base+56	1	12	0000
1	14	0000	Base+58	1	13	0000
0	15	0000	Base+60	1	14	0000
1	15	0000	Base+62	1	15	0000

Figure 7-8 Transforming a Raster from Device-Specific to Standard Format

Remember that the Atari ST screen bit map uses the device-specific format. If you do any copying to and from the screen, be sure to set the format field of the MFDB properly.

After the raster transformations have been demonstrated, the **set_up_rasters( )** function is called next. This function sets up the ~MFDBs for the checkerboard, stripe, and block rasters to be used by the other functions in this program. The addresses, width, height, format, and number of planes are set for each MFDB. Note in

chexMFDB that only one plane is being used at this time. The program simply ignores the second plane. Its only purpose is demonstrating the transformations.

Next the structure **screenMFDB** is set up. The address of the raster (bit map) is obtained by using the **Logbase( )** function, as it was in the BITMAP program. The width and height are initially set to the number of pixels on a high resolution monochrome monitor, and the number of planes is set to one. The screen resolution is tested: the variable **screen_rez** is set in **set_screen_attr( )**. If it is 0, a low resolution screen is being used and the width, height, word width, and number of planes are set appropriately. If a medium resolution screen is being used, the width, height, and number of planes are changed. The word width does not need to be reset under medium resolution because it is the same as for high resolution.

Finally, the temporary MFDB, **tempMFDB**, is set up. The device-specific format is used. The number of planes is set to the same number of planes used by the screen because the temporary raster will store a portion of the screen. Now that all of the MFDBs have been set, the raster test function can be performed.

In **raster_test( )**, the **pxy** array holds the coordinates of the source and destination rectangles. The **color_index** array is used for the transparent copy raster function. The variable **pass** is used to loop through the function twice. The first time the transparent copy raster function is used; the second time the opaque copy raster function is used.

The function begins by setting the two color_index array elements to the foreground color (black) and the background color (white). Inside the loop, the workstation is cleared. A background pattern is drawn by setting the interior fill mode to pattern fill using pattern number 2. Then a filled rectangle is drawn from coordinates (10,10) to (150,150).

The function continues by preparing the **pxy** array to store a portion of the screen into the temporary raster. A 16-by-16 pixel square will be stored from coordinates (100,100) to (115,115). This square is placed in the temporary raster from location (0,0) to (15,15). Remember that the first coordinate of the raster, (0,0), is at the upper left corner of the raster. When saving a portion of the screen, the opaque copy raster function *must* be used because the transparent copy raster function is only for use with a monochrome source raster and a color destination raster. Since you do not know whether your program runs on a system with a monochrome or color screen, it is easiest to use the opaque copy raster function. In this way, if the screen has one, two, or four planes, the opaque copy is able to handle it. The opaque copy raster function is used to copy pixel for pixel from the screen to the temporary raster area. Notice that the second

parameter is the defined constant S_ONLY. This constant, the other 15 logical writing modes for opaque copies, and the four writing modes used by the VDI are defined in the header file obdefs.h. The use of these constants is strongly recommended because they make the program much easier to understand.

At this point, a 16-by-16 square has been copied from the screen into the temporary raster area. Now the checkerboard pattern is drawn into the area on the screen just saved. To do this, the coordinates in the **pxy** array are reversed because the program is copying the image from the checkerboard raster to the screen. Since the whole checkerboard raster is copied, the coordinates are from (0,0) to (15,15). The raster is placed on the screen at screen coordinates (100,100) to (115,115). On the first pass through the loop, the transparent copy raster function with replace mode is used. The program pauses and waits for a key to be pressed. Then the stripe pattern is drawn overlapping the checkerboard pattern slightly. The block pattern is then drawn on the screen at another location.

After another key press, the use of different source and destination rectangle sizes is demonstrated. A 10-by-10 pixel square is taken from the block raster and placed on the screen in a rectangle 40 pixels wide and 10 pixels high. Next the block raster is placed on the screen in a 10-by-40 pixel rectangle.

At the end of the loop after a key press, the program restores the portion of the screen saved earlier. The opaque copy raster function is used to replace that area of the screen pixel for pixel. A pattern was placed on the screen earlier to enable you to see that the image is actually being replaced. If a white background were used, the program would look pretty dull.

On the second pass through the loop, the opaque copy raster function is used instead of the transparent copy raster function. There is a difference between using one type of function or the other depending upon the type of screen you are using. This difference is detailed below.

The next function, **color_test( )**, demonstrates the four writing modes in conjunction with the color index array and the transparent copy raster function. The first thing done is clearing the screen and drawing a colored, filled rectangle. The checkerboard raster is then placed on top of this rectangle using the various writing modes.

Replace mode is used first. The pxy coordinate array and the color index array are set. If a color screen is used, color indices 2 and 3 are used for the foreground and background, respectively; otherwise, black (1) and white (0) are used. The checkerboard raster is copied onto the rectangle using the replace mode. It is then copied using the transparent mode, XOR mode, and reverse transparent mode. When **color_test( )** has finished, the program terminates.

Results from Program RASTER

The output from program RASTER should be studied carefully. This program provides you with a vast amount of information about the way the Atari ST stores and displays screen images.

The first function executed, **raster_form( )**, produces output similar to that shown in Figures 7-7 and 7-8. The figures detail the process of transforming from standard format to device-specific format and from device-specific format to standard format. Remember that standard format consists of contiguous planes with all of one plane followed by all of the next plane. Device-specific format has the words of each plane interleaved.

Monochrome Output

The copy raster functions produce different output depending on the type of display in use. The results presented below refer to output on a monochrome monitor.

The **raster_test( )** function draws a pattern-filled square. Since saving a portion of the screen to memory produces no visible results, the first thing you see is a checkerboard pattern drawn within the pattern-filled square. After you press a key, a stripe pattern is drawn that overlaps the checkerboard slightly. Then the block pattern is shown. After the next key press, you see a rather unusual pattern that is the result of the copying between rectangles of different sizes. Another key press produces another strange pattern. If you get two completely filled rectangles instead of these strange patterns, your version of GEM is working properly. On the author's computer, these copy operations did not work as described in the VDI manual. After another key press, the portion of the screen saved in the temporary raster is restored to the screen.

At this point, the loop repeats and the copy operations use the opaque copy raster function instead of the transparent copy raster function. For a monochrome display, the output of these two functions appears the same.

In the **color_test( )** function, there is only one plane in the monochrome mode, and therefore, only one bit on which the logic operations can be performed. First the black rectangle is drawn. Then the checkerboard raster is drawn using the four writing modes. The replace mode simply places the checkerboard pattern on the screen. The transparent mode makes white areas appear transparent. Thus, the checkerboard is a black and "clear" image on a black background and nothing shows on the screen. The XOR mode gives the reverse image of the pattern. The reverse transparent mode sets all 0 bits to

the background color (white) and has no effect on the 1 bits; the result is the checkerboard pattern itself.

Color Output

The results of the color display are somewhat more interesting. The first function, the **raster_form( )** function, produces the same results as before (see Figures 7-7 and 7-8). In the **raster_test( )** function, the pattern-filled background square is drawn. Then the checkerboard pattern appears, followed by the striped pattern and block pattern. When the program copies rasters with rectangles of different sizes, the results again look similar to the monochrome output. Finally, the area of the screen saved in the temporary raster is restored. So far everything looks the same as the monochrome output. The transparent copy raster function works the same with monochrome or color output.

If you have a monochrome monitor and a color monitor (or television), you may notice that the pattern used to fill the background square looks different on a color monitor. This is probably due to the lower resolution of the color monitor (or television) and not due to the copy raster function.

At this point, the program is ready for the second pass through the loop. Now the program uses the opaque copy raster function. After the pattern-filled square is drawn, you see no changes. Press a key; still there are no results. In fact, nothing shows on the screen during this pass through the loop. The opaque copy function does not work when copying from a monochrome raster to a color raster. In this case, the program is trying to copy a one-plane monochrome image onto a four-plane color image. Basically, the VDI does not perform an opaque copy between rasters that have a different number of planes. This makes sense because an opaque copy is done pixel for pixel; if there aren't enough pixels to supply the number of planes being copied to or from, this type of operation doesn't make sense. The VDI recognizes this situation and ignores the function call.

After the **raster_test( )** function is completed, the program moves on to the **color_test( )** function. On the color monitor, it is again a little more interesting than on the monochrome monitor. The function draws a colored background rectangle and places the four checkerboard patterns on the rectangle. The actual colors shown on the screen vary depending on how the color palette is set on your computer. Four different colors are shown based upon the four different writing modes used. As an exercise, you should work out the logic operations of the four writing modes between the color of the background rectangle and the color used in the **color_index** array. You should then be able to predict which colors will be shown for each writing mode.

Playing with Program RASTER

If you are using a color monitor, you should set up a raster with four planes and test the opaque copy function. This should give you some results on the second pass through **raster_test( )**. Another exercise would be to change the function to store the portion of the screen into the temporary raster area from opaque to a transparent copy raster. In this case, the screen is not restored because the transparent copy raster function is designed to copy from a monochrome raster to a color raster. Unless a monochrome screen is used, the VDI ignores the function call.

Go ahead and make some of the changes suggested. Try other experiments such as different sized rasters, making rasters with more planes, and so on. Remember that the opaque raster function must use the device-specific format. If it is easier for you to design the raster in standard format, go ahead and do that and then use the transformation function to transform the raster into device-specific format.

Putting It All Together: Program BOUNCE

The program BOUNCE is a culmination of all of the VDI routines and extended BIOS routines you've seen so far. It uses rasters and the screen bit map techniques to show four small "balls" bouncing up and down and across the screen.

The balls start in the upper left corner of the screen. Each is given an initial horizontal velocity and falls under a constant acceleration (gravity). When a ball hits the bottom of the screen, it bounces back up. When a ball hits the side of the screen it bounces off the side and heads in the opposite direction.

Putting first things first, as always, we take a look at the data and data structures to be used in the program. There is the usual GEM application overhead including the MFDB structure declaration. Under the application-specific data there are three defined constants: W_BALL, H_BALL, and NUM_BALLS. W_BALL and H_BALL define the width and height in pixels of the ball raster to be used. NUM_BALLS determines the number of balls to appear on the screen.

Because each ball is allowed to move independently, it is in a unique state at any given time. In particular, each ball can have a different position on the screen and a different vertical and horizontal velocity. To keep track of this information in an organized fashion, the structure BALL_REC is declared in the application-specific data section. In this structure, there are two x coordinates, two y coordinates, an x velocity, a y velocity, and a delay. There are two sets of

coordinates because this is going to be an animated program like ANIMATE. Therefore, two different bit maps are used to display the moving objects. With two bit maps, there must be two sets of coordinates—one for each bit map. The integer element delay is used to determine when each ball is released. This allows the balls to start moving independently.

Two pointers to the screen bit maps, bitmap0 and bitmap1, are declared next. These are followed by two integers, **draw_screen** and **show_screen**. Variable **draw_screen** indicates which screen (0 or 1) is currently being drawn on. Variable **show_screen** indicates which screen is currently visible. If **draw_screen** is 1, then **show_screen** must be 0; if **draw_screen** is 0, **show_screen** must be 1. The array **ball** is declared to have as many elements as there are balls (as specified by NUM_BALLS). The **float** variable **gravity** determines how the balls are to be accelerated.

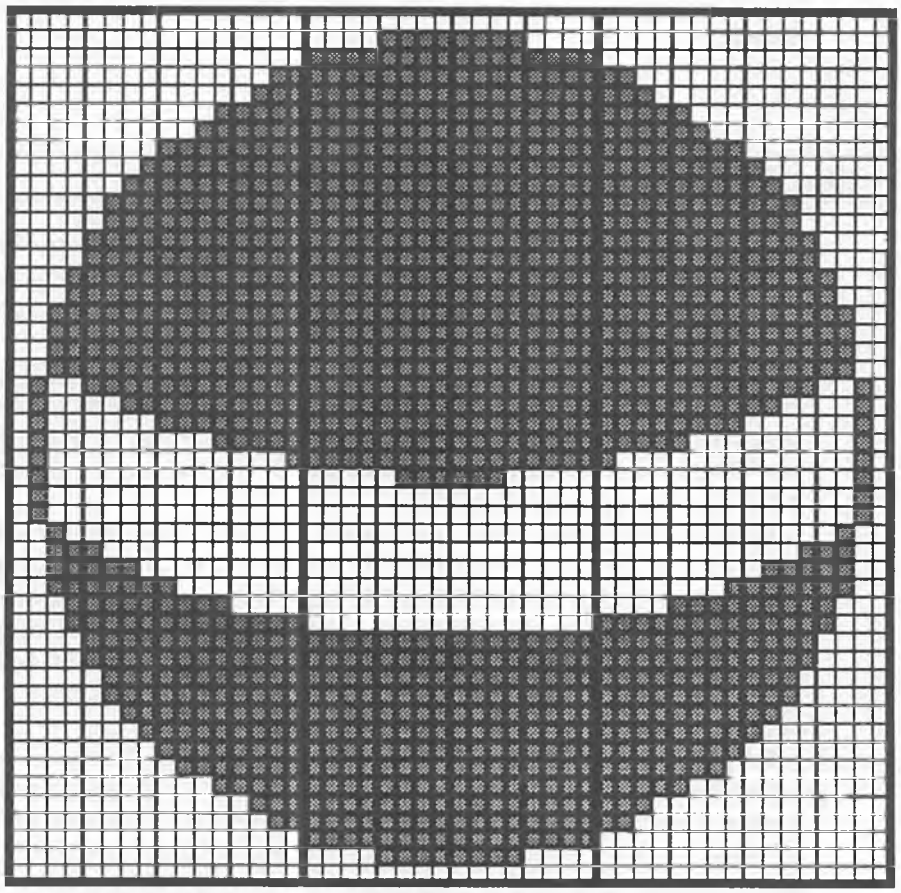


Figure 7-9 Ball Raster

There are four memory form definition blocks (MFDBs) declared: one for the ball, one for its mask, and one for each screen. Figure 7-9 shows the raster for the ball. The grid in the figure is 48 boxes by 48 boxes, corresponding to a 48-by-48 pixel raster. Just as with the pattern rasters in program RASTER, the image is divided into 4-pixel horizontal groups, which are translated into hexadecimal notation (see Figure 7-9). Another advantage of using rasters is that the raster image can be changed quite easily. You may want to change the ball to have a different pattern on its face or look like something completely different such as a pogo stick.

Once the hexadecimal codes for the ball raster have been determined, they must be stored in memory so that the VDI can use them as a raster. The raster is held in array **ball_shape**. Note that the array's dimensions are 48 by 3. Looking at Figure 7-9, you can see two heavy vertical lines that divide the raster into three 16-bit groups and that there are 48 lines (rows) vertically. These physical dimensions determine the dimensions of the array. The raster is three words wide (16 bits per word) and 48 rows high.

Consider the following animation technique. The ball is drawn at a certain point on the screen with all dark areas of the ball denoted by bits with a value of 1 and all white areas denoted by bits with a value of 0. To make the ball appear to move, it is first redrawn in the same location using the XOR writing mode, which in effect erases the image. The ball is then drawn in its new position using the normal replace mode (or XOR mode again). The new image is erased and the next image is drawn in a new location. By repeating this quickly enough, the ball appears to move.

This program, however, uses not one but four balls on the screen at one time. Since the balls will cross paths, the program must be able to draw the balls so that they appear to be on top of each other. Whenever one ball is drawn on top of the other, it should appear as if that ball is in front of the other. In other words, the top ball should partially or totally mask the other ball or balls it covers. You can't use replace mode because the raster is square; the portion of the raster around the ball would also be drawn on the screen and you would get a shadow around the ball. You can't use transparent mode because anything beneath the ball would show through the white stripe. Reverse transparent mode would draw the ball incorrectly. If the XOR writing mode were used, a "checkerboard" pattern would appear where the balls overlap. This means that any place a black portion is drawn over a black portion appears white and any place a white portion is drawn over white appears black. Therefore, this would not give the proper image.

A method is needed that erases the portion of the screen under the topmost ball so nothing shows through the topmost ball. This is

accomplished through the use of a *mask*. A mask is a raster that has all its bits set to 1 wherever the image should appear solid. If you use a mask, you can draw over anything using transparent mode so that the square edges do not show and erase the area around the image. Then draw the actual image. The mask for the ball in this program covers any area of the raster covered by the ball. Thus, the mask is just a filled circle (essentially the ball itself without the white stripe in the middle). The mask raster is initialized in the array **ball_mask**. With the help of graph paper, you can verify that the values in **ball_mask** correspond to such a filled-in circle. The area in the mask not covered by the ball has its bits set to 0.

Other examples of a mask are readily apparent. On the desktop, for example, the disk icons and the trash can icons are held in memory as rasters and have corresponding masks. The masks look like shadows of the icons. The mouse cursor also has a mask, which is most noticeable when it looks like a finger. The finger is filled with white so a mask is used to make the inside of the hand appear solid.

Operation of Program BOUNCE

Program BOUNCE contains most of the graphic techniques shown in the preceding chapters. Rasters, multiple screens, and masks are used to show the bouncing balls. At the top of the program, the usual GEM routines initialize access to the workstation. In the application-specific routines, the **set_bitmap( )** function is a modified version of the **set_base( )** function used in ANIMATE. This routine returns the address of the current screen bit map and sets the parameter to the address of a new screen bit map. This allows any future programs to use any variable name you want for the bit map addresses. The function **set_up_rasters( )** sets up the rasters for the ball, the mask, and both screens.

In the general control flow of the program in **main( )**, the application-specific functions start by initializing the bit map addresses and the raster MFDBs (see Listing 7-2). The program then enters a loop that controls the execution of the program from this point on. First, the display screen and drawing screen are set and cleared. Then the ball array is initialized in **init_ball( )** and function **bounce_ball( )** is called to start the animation sequence. Function **bounce_ball( )** returns when a key has been pressed. Upon its return, the loop in **main( )** checks if the key pressed is the **ESC** key. If the **ESC** key is pressed, the program is exited; otherwise, the program starts the balls bouncing again. On exiting, the program makes sure the screen is set to the original bit map, the secondary bit map memory is released, and the virtual workstation is closed.

Listing 7-2 Program BOUNCE

```
/*********************************************************************
     BOUNCE.C Draw bouncing balls

     This program demonstrates an animation technique using
     raster operations.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS rowtines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

typedef   struct    mfdbstr        /* GEM data structure */
{
     char  *addr;                  /* address of raster area */
     WORD  wide;                   /* width of raster in pixels */
     WORD  high;                   /* height of raster in pixels */
     WORD  word_width;             /* width of raster in words */
     WORD  format;                 /* standard or device specific */
     WORD  planes;                 /* number of planes in raster */
     WORD  reserv1, reserv2, reserv3;
} MFDB;

/*************************************************
     Application Specific Data
*************************************************/
```

Listing 7-2 (continued)

```
#define   W_BALL    48          /* width of ball raster */
#define   H_BALL    48          /* height of ball raster */
#define   NUM_BALLS  4          /* number of balls to draw */

typedef   struct
{
     int       x[2];            /* x coord for each screen */
     int       y[2];            /* y coord for each screen */
     float     dx;              /* x velocity */
     float     dy;              /* y velocity */
     int       delay;           /* release delay timer */
} BALL_REC;

char *bitmap0,                  /* screen bitmaps */
     *bitmap1;

int draw_screen,                /* 0 or 1; determines which */
     show_screen;               /* screen to use for drawing */

BALL_REC  ball[NUM_BALLS];      /* a record for each ball */

float     gravity;              /* acceleration */

MFDB ballMFDB,                  /* raster descriptors */
     maskMFDB,
     scrMFDB[2];                /* one for each screen */

/* define shapes */
int  ball_shape[48][3] = {      /* Line */
     0x0000, 0x0000, 0x0000,    /* 0 */
     0x0000, 0x0ff0, 0x0000,
     0x0000, 0xffff, 0x0000,
     0x0003, 0xffff, 0xc000,
     0x0007, 0xffff, 0xe000,
     0x001f, 0xffff, 0xf800,
     0x007f, 0xffff, 0xfe00,
     0x00ff, 0xffff, 0xff00,
     0x01ff, 0xffff, 0xff80,    /* 8 */
     0x03ff, 0xffff, 0xffc0,
     0x07ff, 0xffff, 0xffe0,
     0x07ff, 0xffff, 0xffe0,
     0x0fff, 0xffff, 0xfff0,
     0x0fff, 0xffff, 0xfff0,
     0x1fff, 0xffff, 0xfff8,
     0x1fff, 0xffff, 0xfff8,
     0x3fff, 0xffff, 0xfffc,    /* 16 */
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x4fff, 0xffff, 0xfff2,
     0x43ff, 0xffff, 0xffc2,
```

Listing 7-2 (continued)

```
     0x40ff, 0xffff, 0xff02,
     0x401f, 0xffff, 0xf802,
     0x4001, 0xffff, 0x8002,        /* 24 */
     0x4000, 0x07a0, 0x0002,
     0x4000, 0x0000, 0x0002,
     0x4000, 0x0000, 0x0002,
     0x2000, 0x0000, 0x0004,
     0x3800, 0x0000, 0x001c,
     0x3a00, 0x0000, 0x007c,
     0x3f80, 0x0000, 0x01fc,
     0x1ff0, 0x0000, 0x0ff8,        /* 32 */
     0x1fff, 0x0000, 0xfff8,
     0x0fff, 0xffff, 0xfff0,
     0x0fff, 0xffff, 0xfff0,
     0x07ff, 0xffff, 0xffa0,
     0x07ff, 0xffff, 0xffa0,
     0x03ff, 0xffff, 0xffc0,
     0x01ff, 0xffff, 0xff80,
     0x00ff, 0xffff, 0xff00,        /* 40 */
     0x007f, 0xffff, 0xfa00,
     0x001f, 0xffff, 0xf800,
     0x0007, 0xffff, 0xa000,
     0x0003, 0xffff, 0xc000,
     0x0000, 0xffff, 0x0000,
     0x0000, 0x0ff0, 0x0000,
     0x0000, 0x0000, 0x0000
     };

int  ball_mask[48][3] = {          /* Line */
     0x0000, 0x0000, 0x0000,        /* 0 */
     0x0000, 0x0ff0, 0x0000,
     0x0000, 0xffff, 0x0000,
     0x0003, 0xffff, 0xc000,
     0x0007, 0xffff, 0xa000,
     0x001f, 0xffff, 0xf800,
     0x007f, 0xffff, 0xfa00,
     0x00ff, 0xffff, 0xff00,
     0x01ff, 0xffff, 0xff80,        /* 8 */
     0x03ff, 0xffff, 0xffc0,
     0x07ff, 0xffff, 0xffa0,
     0x07ff, 0xffff, 0xffa0,
     0x0fff, 0xffff, 0xfff0,
     0x0fff, 0xffff, 0xfff0,
     0x1fff, 0xffff, 0xfff8,
     0x1fff, 0xffff, 0xfff8,
     0x3fff, 0xffff, 0xfffc,        /* 16 */
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x7fff, 0xffff, 0xfffa,
     0x7fff, 0xffff, 0xfffa,
     0x7fff, 0xffff, 0xfffa,
```

Listing 7-2 (continued)

```
     0x7fff, 0xffff, 0xfffe,
     0x7fff, 0xffff, 0xfffe,        /* 24 */
     0x7fff, 0xffff, 0xfffe,
     0x7fff, 0xffff, 0xfffe,
     0x7fff, 0xffff, 0xfffe,
     0x3fff, 0xffff, 0xfffe,
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x3fff, 0xffff, 0xfffc,
     0x1fff, 0xffff, 0xfff8,        /* 32 */
     0x1fff, 0xffff, 0xfff8,
     0x0fff, 0xffff, 0xfff0,
     0x0fff, 0xffff, 0xfff0,
     0x07ff, 0xffff, 0xffe0,
     0x07ff, 0xffff, 0xffe0,
     0x03ff, 0xffff, 0xffc0,
     0x01ff, 0xffff, 0xff80,
     0x00ff, 0xffff, 0xff00,        /* 40 */
     0x007f, 0xffff, 0xfe00,
     0x001f, 0xffff, 0xf800,
     0x0007, 0xffff, 0xe000,
     0x0003, 0xffff, 0xc000,
     0x0000, 0xffff, 0x0000,
     0x0000, 0x0ff0, 0x0000,
     0x0000, 0x0000, 0x0000
     };

/***********************************************
     GEM-related Functions
***********************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/***********************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
***********************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;               /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)       /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;               /* use raster coords */
     new_handle = phys_handle;      /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}
```

Listing 7-2 (continued)

```
set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/


long Rnd_rng(low, hi)
long low, hi;
/*************************************************
Function: Generate a random number between low and hi, inclusive.
Input:    low = lowest value in range.
          hi  = highest value in range.
Output:   Returns random number.
*************************************************/
{
     hi++;                         /* include hi value in range */
     return( (Random() % (hi - low)) + low);
}

char *set_bitmap(new_map)
char **new_map;
/*************************************************
Function: Allocate memory for new screen bitmap.
Input:    new_map = pointer to a pointer for the new bitmap.
Output:   Returns current bitmap address and sets new_map
          to address of new bitmap location.
*************************************************/
{
long x;

/* allocate new screen bitmap */
     x = (long)Malloc(32256L);     /* get 32 kbytes */
     if (!(x % 256))               /* on half page boundary */
          *new_map = (char *) x;
     else                          /* move to boundary */
          *new_map = (char *)(x + (256 - (x % 256)));
```

Listing 7-2 (continued)

```
/* return current bitmap */
     return( (char *)Logbase() );
}

set_up_rasters()
/***********************************************
Function: Sets up global MFDB to point to rasters.
Input:    None. Uses global FDBs and rasters listed above.
Output:   Sets appropriate fields in FDBs.
***********************************************/
{

/* Ball raster */
     ballMFDB.addr       =    (char *)ball_shape;
     ballMFDB.wide       =    W_BALL;    /* width in pixels */
     ballMFDB.high       =    H_BALL;    /* height in pixels */
     ballMFDB.word_width =    3;         /* width in WORDs */
     ballMFDB.format     =    0;
     ballMFDB.planes     =    1;         /* monochrome has 1 plane */

/* Mask raster */
     maskMFDB.addr       =    (char *)ball_mask;
     maskMFDB.wide       =    W_BALL;    /* width in pixels */
     maskMFDB.high       =    H_BALL;    /* height in pixels */
     maskMFDB.word_width =    3;         /* width in WORDs */
     maskMFDB.format     =    0;
     maskMFDB.planes     =    1;         /* monochrome has 1 plane */

/* Screen raster area */
     scrMFDB[0].wide          =    640;
     scrMFDB[0].high          =    400;
     scrMFDB[0].word_width    =    40;
     scrMFDB[0].format        =    0;
     scrMFDB[0].planes        =    1;     /* monochrome has 1 plane */
     if (screen_rez == 0)                 /* low resolution */
     {
          scrMFDB[0].wide     =    320;
          scrMFDB[0].high     =    200;
          scrMFDB[0].word_width = 20;
          scrMFDB[0].planes   =    4;
     }
     else if (screen_rez == 1)            /* medium resolution */
     {
          scrMFDB[0].wide     =    640;
          scrMFDB[0].high     =    200;
          scrMFDB[0].planes   =    2;
     }

/* second bitmap sams as first */
     scrMFDB[1].wide          =    scrMFDB[0].wide;
     scrMFDB[1].high          =    scrMFDB[0].high;
```

Listing 7-2 (continued)

```
     scrMFDB[1].word_width    =    scrMFDB[0].word_width;
     scrMFDB[1].format        =    scrMFDB[0].format;
     scrMFDB[1].planes        =    scrMFDB[0].planes;

     scrMFDB[0].addr     =    bitmap0;
     scrMFDB[1].addr     =    bitmap1;

     return;
}

init_ball()
/*************************************************
Function: Initialize ball variables.
Input:    None.
Output:   Set ball position, movement, and speed.
*************************************************/
{
int  i;
WORD work_out[57];

                                /* get distance between pixels */
     vq_extnd(screen_vhandle, 0, work_out);
                                /* determine acceleration factor */
     gravity = 9800000 / ( 3600 * (float)work_out[4]);

     for (i = 0; i < NUM_BALLS; i++)
     {
          ball[i].x[0]  = ball[i].x[1] = 0;
          ball[i].y[0]  = ball[i].y[1] = 0;
          ball[i].dx  = Rnd_rng(1L, 10L);     /* x movement */
          ball[i].dy  = 0;                    /* start at top of arc */
          ball[i].delay = Rnd_rng(1L, 100L);
     }
}

draw_ball(x, y)
WORD x, y;
/*************************************************
Function: Draw a ball at the specified coordinates.
Input:    x    = x coord of upper left corner.
          y    = y coord of upper left corner.
          Raster must be set.
Output:   None.
*************************************************/
{
WORD pxy[8],
     color_index[2];

     color_index[0] = 1;           /* Foreground color value */
     color_index[1] = 0;           /* Background color value */
```

Listing 7-2 (continued)

```
/* Set coordinate array */
     pxy[0] = 0;                    /* source x1 coord */
     pxy[1] = 0;                    /* source y1 coord */
     pxy[2] = W_BALL - 1;           /* source x2 coord */
     pxy[3] = H_BALL - 1;           /* source y2 coord */
     pxy[4] = x;                    /* dest x1 coord */
     pxy[5] = y;                    /* dest y1 coord */
     pxy[6] = x + pxy[2];           /* dest x2 coord */
     pxy[7] = y + pxy[3];           /* dest y2 coord */

/* use mask to erase area under ball */
     color_index[0] = 0;       /* 1 bits set to background color */
     vrt_cpyfm(screen_vhandle, MD_TRANS, pxy,
          &maskMFDB, &scrMFDB[draw_screen], color_index);

/* draw ball */
     color_index[0] = 1;       /* 1 bits set to foreground color */
     vrt_cpyfm(screen_vhandle, MD_TRANS, pxy,
          &ballMFDB, &scrMFDB[draw_screen], color_index);
}

erase_ball(x, y)
WORD x, y;
/*************************************************
Function: Draw a ball at the specified coordinates.
Input:    x    = x coord of upper left corner.
          y    = y coord of upper left corner.
          Rasters must be set.
Output:   None.
*************************************************/
{
WORD pxy[8],
     color_index[2];

     color_index[0] = 0;            /* Foreground color value */
     color_index[1] = 0;            /* Background color value */

/* Set coordinate array */
     pxy[0] = 0;                    /* source x1 coord */
     pxy[1] = 0;                    /* source y1 coord */
     pxy[2] = W_BALL - 1;           /* source x2 coord */
     pxy[3] = H_BALL - 1;           /* source y2 coord */
     pxy[4] = x;                    /* dest x1 coord */
     pxy[5] = y;                    /* dest y1 coord */
     pxy[6] = x + pxy[2];           /* dest x2 coord */
     pxy[7] = y + pxy[3];           /* dest y2 coord */

/* use mask to erase area under ball */
     color_index[0] = 0;  /* 1 bits set to background color */
     vrt_cpyfm(screen_vhandle, MD_TRANS, pxy,
          &maskMFDB, &scrMFDB[draw_screen], color_index);
}
```

Listing 7-2 (continued)

```
calc_ball()
/*************************************************
Function: Calculate new position for balls.
Input:    Uses ball position variables.
Output:   New ball position, acceleration, and direction.
*************************************************/
{
int  i;


for  (i = 0; i < NUM_BALLS; i++)
{
     ball[i].delay--;              /* count down */
     if (ball[i].delay > 0)        /* do not release ball */
          continue;


     ball[i].dy += gravity;        /* change velocity */
     /* change position */
     ball[i].y[draw_screen] = ball[i].y[show_screen] + ball[i].dy;
     ball[i].x[draw_screen] = ball[i].x[show_screen] + ball[i].dx;


/* check range */
     /* left edge */
     if (ball[i].x[draw_screen] < 0)
     {
          ball[i].x[draw_screen] = 0;
          ball[i].dx = -ball[i].dx;
     }
     /* right edge less width of ball */
     if (ball[i].x[draw_screen] > (x_max - W_BALL))
     {
          ball[i].x[draw_screen] = x_max - W_BALL;
          ball[i].dx = -ball[i].dx;
     }


     /* top of screen */
     if (ball[i].y[draw_screen] < 0)
     {
          ball[i].y[draw_screen] = 0;
          ball[i].dy = 0;
     }
     /* bottom of screen */
     if (ball[i].y[draw_screen] > (y_max - H_BALL))
     {         /* at bottom of arc ball bounces losing energy */
          ball[i].y[draw_screen] = y_max - H_BALL;
          ball[i].dy = -ball[i].dy * 0.975;
     }
}
return;
}
```

Listing 7-2 (continued)

```
bounce_ball()
/*************************************************
Function: Bounce balls on screen.
Input:    None.
Output:   None.
*************************************************/

{
int  i;

     do
     {                              /* draw balls on hidden screen */
          for (i = 0; i < NUM_BALLS; i++)
               draw_ball(ball[i].x[draw_screen], ball[i].y[draw_screen]);
                                    /* switch screens */
          if (draw_screen)
          {
               draw_screen = 0;
               show_screen = 1;
          }
          else
          {
               draw_screen = 1;
               show_screen = 0;
          }
          Setscreen(scrMFDB[draw_screen].addr,
               scrMFDB[show_screen].addr, -1L);
          Vsync();
                                    /* erase balls */
          for (i = 0; i < NUM_BALLS; i++)
               erase_ball(ball[i].x[draw_screen], ball[i].y[draw_screen]);
                                    /* calc new ball positions */
          calc_ball();
     } while (!Cconis());
}

/*************************************************
     Main Program
*************************************************/

main()
{
int ap_id;                          /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/
```

Listing 7-2 (continued)

```
    ap_id = appl_init();          /* Initialize AES routines */
    if (ap_id < 0)                /* no calls can be made to AES */
    {                             /* use GEMDOS */
        Cconws("***> Initialization Error. <***\n");
        Cconws("Press any key to continue.\n");
        Crawcin();
        exit(-1);                 /* set exit value to show error */
    }

    screen_phandle =              /* Get handle for screen */
        graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
    screen_vhandle = open_vwork(screen_phandle);
    set_screen_attr();            /* Get screen attributes */

/**************************************************
    Rpplication Specific Routines
**************************************************/

    bitmap0 = set_bitmap(&bitmap1);
    set_up_rasters();
    do
    {
        show_screen = 0;              /* set to current screen */
        draw_screen = 1;              /* set to background screen */
        Setscreen(bitmap1, -1L, -1L); /* clear screens */
        v_clrwk(screen_vhandle);
        Setscreen(bitmap0, -1L, -1L);
        v_clrwk(screen_vhandle);
        init_ball();
        bounce_ball();                /* exits on keypress */
    } while ( (Crawcin() & 0x7F) != 27);

/**************************************************
    Program Clean-up and Exit
**************************************************/

    Setscreen(bitmap0, bitmap0, -1L);
    Mfree(bitmap1);               /* return secondary bitmap */
    v_clsvwk(screen_vhandle);     /* close workstation */
    appl_exit();                  /* end program */
}
/**************************************************/
```

As for the application-specific functions of the program, the **Rnd_rng( )** range function and the **set_bitmap( )** function have each been seen before. The **set_up_rasters( )** function sets the MFDBs of a monochrome raster in device-specific format for the ball and mask. These are both three words wide. An MFDB for each display screen

is also set up. The only difference between the display screens is their address.

Function **init_ball( )** performs an initialization of the ball records. The first thing the function does is to make an extended inquiry requesting the open workstation values. Element 4 of the array **work_out** is a value that indicates the distance between the top of one pixel and the top of the next pixel. This distance is measured in microns (1 micrometer = 1/1,000,000 meter) and is used to determine the value for gravity. Gravity is computed in terms of pixels per frame. A frame is considered to be 1/60 of a second—the refresh cycle time. After gravity has been calculated, each ball record is then set to its initial values. All balls start in the upper left corner. An x velocity is selected randomly, and the y velocity is set at 0. The delay is also randomly chosen so that the balls are released at different times.

Function **draw_ball( )** draws a ball at the given x and y coordinates. These coordinates specify the upper left corner of the source raster on the destination raster. As usual, the arrays **color_index** and **pxy** are required. The color 1 is used as a foreground color and the background color is set to 0. The **pxy** array has its first four coordinates set to the size of the source raster, which has opposite corners at (0,0) and (47,47). The destination coordinates are then set. The upper left coordinates are set at (x,y) and the lower right coordinates are determined and set by adding on the size of the source raster.

Since the area under the ball must be erased, the mask is drawn first. The foreground color is set to the background color and the transparent copy raster function is used. Remember that the transparent copy raster function works with both color and monochrome screens. The function draws with the transparent mode to copy the mask to the screen. The transparent mode is used because any place there is a 0 in the mask raster, the screen is not changed. Any place there is a 1 in the mask raster, the background color is drawn.

Once the mask has been drawn, the color is set back to the foreground color and the ball raster is drawn on the area just erased. Again the transparent mode is used: where there is a 0 in the raster, no action takes place.

The next function, **erase_ball( )**, erases the ball from the screen at coordinate (x,y). This function is the same as the **draw_ball( )** function except that the **erase_ball( )** function stops after erasing the screen.

Calc_ball( ) calculates the new position of each ball in a loop. This function is called for each new frame (refresh cycle). The delay value for each ball is decremented and tested. If the value is still positive, the delay has not been completed and the loop continues. If the ball is ready to be released or has already been released, the y velocity is increased. The new position is calculated based upon the velocity and

the x and y values of the ball's current position (the position on the screen now being shown). The new x and y positions are held in the x and y values for the screen used for drawing. If the ball is to be drawn at the next position, the calculation must obviously take into account the present position and the present velocity.

Once the new position of the ball is determined, the final step is to check if the ball has gone beyond the edges of the screen. If the ball has gone past the left edge, the x position is set to 0 and the x velocity of the ball is reversed. If the ball goes past the right edge (adjusted for the width of the ball), the position is set to the right edge (minus the ball's width) and the x velocity is reversed. If the ball goes above the top of the screen (which should not occur but should be tested for safety), the y position is set to the top of the screen and the y velocity becomes 0 (the top of the arc). When the ball hits the bottom of the screen, it should bounce. The y value is set to the bottom of the screen (adjusted for the height of the ball) and the ball's y velocity is reversed and decreased slightly. In nature, a rubber ball does not continue to bounce forever. When a ball strikes the ground, it bounces up at a slightly slower velocity than when it hit. In this program, the upward speed is set to 97.5% of the downward speed.

Function **bounce_ball( )** is simply a loop. First it draws all balls on the drawing screen. Next the screens are swapped and a pause for vertical synchronization is made. Then the balls are erased from the new drawing screen (the one that has just been swapped from being visible) and the new ball positions are calculated. When the user presses a key, the loop ends and **bounce_ball( )** returns to function **main( )**.

Note that the screen is always drawn in its entirety. In other words, the four balls are erased from the screen by **erase_ball( )**, which makes the screen blank. Then four new balls are drawn on the screen. This is the basic flow for almost any type of animation program even if you want to have a complex background on the screen. The primary objective when writing an animation program is to do all drawing as fast as possible to reduce the amount of flicker. When you have a background—even a simple one, it is time-consuming to erase the screen, redraw the background, and then draw the shapes (in BOUNCE the shapes are the balls). To solve this problem, you start with the background. Now assume you are drawing balls on the background. Each time you draw new balls, you first save the area under each ball to be drawn in a temporary raster, similar to the procedure used in program RASTER. Draw the balls using the mask and shape raster.

To erase the shapes, instead of erasing the balls from the screen, simply replace the saved portion of the screen. This writes over the shape and restores the background at the same time. You must save

the rasters before anything is drawn on the screen; otherwise, only part of a ball may be saved. In other words, the rasters are saved first, the balls are drawn, and then the rasters are replaced. If you want to try this, fill in the background of the screen and make the necessary changes to the program. Enter program BOUNCE into your system, and check it out.

Say "Good-bye" to the VDI

At this point, most of the VDI functions have been covered. In this chapter, you learned about one of the more complex and powerful concepts of the VDI—the raster. With the raster, you are able to efficiently write animated programs and provide fast graphic responses within your programs through the use of icons and pictures.

The next chapter deals with one of the interesting features of the Atari ST: sound. After exploring the sound capabilities of the ST, the book moves on to the AES, the second portion of GEM. From this point on, the programs you see will have very few VDI functions in them. You should understand the VDI and how it works. Now move on to making some noise.

CHAPTER EIGHT

Sound Off!

So far you have gone through the VDI and looked at all its features. Now we pause from GEM and look at some of the features of the Atari computer itself. The primary focus of this chapter is the sound output capabilities of the Atari. The chapter starts with some background on sound and how it is generated. Then some sound circuitry and machine capabilities are discussed. Next you look at some components used to create sound on the Atari. Finally, you develop a sound program that demonstrates the various capabilities of the machine.

What Is Sound?

Sound is a wave. From the physics point of view, it is called a *mechanical wave*, which means that some physical object is vibrating. For example, sound waves cause small pulsations in the air. When the pulsations move through the air and reach your ear, they cause your eardrum to vibrate as well. The brain receives the signals from the ear and interprets them as sound.

A *waveform* looks very much like the waves on the open ocean. Figure 8-1 depicts a sample waveform with the vertical axis measuring the *amplitude* of the wave and the horizontal axis measuring time. Note that several attributes of the wave have been labeled. The *period*, or *cycle*, is the time it takes for the wave to complete one repetition. The period measured in Figure 8-1 shows the wave starting at amplitude 0 with a positive slope (that is, a slope that moves

upward). The wave goes up and then down back to amplitude 0 (but now with a negative slope) and goes down and back up to complete the cycle.

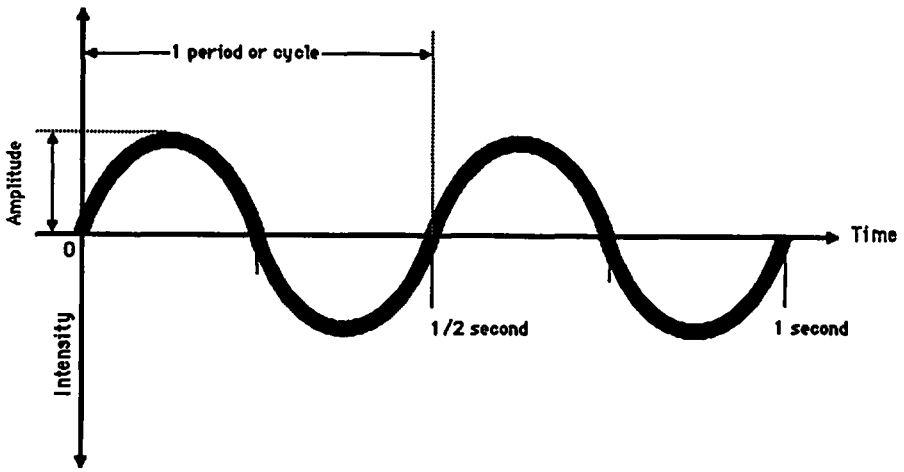


Figure 8-1 Waveform Components

Amplitude measures how high the wave travels above the center line. In general, waves with greater amplitude contain a greater amount of power. We all know that large ocean waves can certainly do more damage than small ones! In relation to sound, the same rule applies: loud noises have a greater amplitude than soft noises.

Frequency is another important component of the wave. It is usually measured in cycles per second (also called hertz, abbreviated as Hz). The frequency measures the number of cycles completed in a given time. In the waveform shown in Figure 8-1, the wave goes through two cycles in one second. Therefore, its frequency is two hertz. Note the relationship between the period and the frequency: the period is the number of seconds per cycle while the frequency is the number of cycles per second. As the units suggest, the period is simply the inverse of the frequency and vice-versa. Thus, if the frequency is very high, you know that there must be many periods each second, which means that the period is very small. In the case of a wave with a frequency of 1000 hertz, for example, each cycle has a period of one 1/1000th of a second.

How does this relate to sound on the Atari ST? The only way the Atari can produce sound is through a speaker, which produces sound by vibrating. The rate at which the speaker vibrates determines the frequency of the sound. Increased frequency corresponds to increased pitch. For example, the note A above middle C has a frequency of 440 hertz. This is the note used by concert orchestras for tuning the

instruments. The next octave above occurs at double that frequency, or 880 hertz. The computer needs some mechanism to drive the speaker at whatever frequency is desired.

Making the Circuit

The electric sound circuitry in the Atari ST is based on the AY-3-8910 Programmable Sound Generator (PSG) made by the General Instrument Corporation. This microchip interfaces with the rest of the computer and allows you to specify frequencies and volume in a program, and generates the output to vibrate the speaker. The PSG can produce a different sound on three independent voices. It is capable of producing either noise (like static) or a tone (like musical notes). You can select any voice and any tone-noise combination you desire. You can have any one voice producing a tone, or noise, or both. Each voice has independent volume control, which can either be set at a constant level or fluctuate in a pattern.

The PSG has 16 registers numbered 0 through 15. Each register controls a particular sound-producing attribute (see Figure 8-2). Registers 0 and 1 control the frequency of voice A, registers 2 and 3 control the frequency for voice B, and registers 4 and 5 control the frequency for voice C. Register 6 is the noise period. Register 7 is the voice enable register, which selects which voices are on and the type of output for each voice. Registers 8, 9, and 10 control the volumes for voices A, B, and C, respectively. Registers 11, 12, and 13 control the envelope period and type of envelope for the volume pattern. Registers 14 and 15 are input and output ports between the PSG and other components of the system. Since the Atari ST uses these ports for sensitive system operations, we do not deal with them.

Setting the Voice Period Registers

The first six registers control the frequency of the output for the three voices. Each voice requires two registers to hold the value that selects a frequency. The actual value held in these registers is not the frequency but rather the period of the tone. The period consists of a 12-bit value, where the four high bits numbered 8 through 11 go in the *coarse tune* register for the voice, and the eight low bits go into the *fine tune* register (see Figure 8-2). (Remember that bit numbering starts at 0 and goes from right to left.) To determine the period of a frequency, convert the frequency into a period, and then split the period into the two register values. You know from the discussion above that the period is the inverse of the frequency. That's fine for

Register	Use	Bit Number b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	Voice A Tone Period	Fine Tune							
1						Coarse Tune			
2	Voice B Tone Period	Fine Tune							
3						Coarse Tune			
4	Voice C Tone Period	Fine Tune							
5						Coarse Tune			
6	Noise Period				5-bit value				
7	Voice $\overline{\text{Enable}}$	$\overline{\text{In}}$/Out		$\overline{\text{Noise}}$			$\overline{\text{Tone}}$		
		IOB	IOA	C	B	A	C	B	A
8	Voice A Volume				Env	4-bit value			
9	Voice B Volume				Env	4-bit value			
10	Voice C Volume				Env	4-bit value			
11	Envelope Period	Fine Period Value							
12		Coarse Period Value							
13	Envelope Shape/Cycle					4-bit Cycle			
14	I/O Port A Data	8-bit data value							
15	I/O Port B Data	8-bit data value							

Figure 8-2 General Instrument AY-3-8910 Registers

humans, but the PSG needs an adjusted period value because the timing of the chip is driven by an electrical signal called the *clock input*. The clock input is supplied at a constant frequency, and the output frequency you want heard must be adjusted by the frequency of that clock input. The frequency of the clock input is 2 MHz, or 2 million cycles per second. This frequency is used to synchronize all the components in the system. The sound chip uses 1/16 of the clock input so that you must divide 2 MHz by 16.

Given a particular output frequency (f), you can determine the period (p) for this frequency by the following equation:

$$p = 1 / f \qquad (1)$$

To determine the *tone period* (TP) used by the PSG, simply multiply the period from equation 1 by the clock adjustment.

$$TP = p * (2\ \text{MHz} / 16) \qquad (2)$$

By combining the two preceding equations you get a single equation that given the frequency yields the tone period for the PSG tone registers to use. This equation is as follows:

$$TP = 2\ MHz\ /\ (16 * f) \qquad (3)$$

For example, if you want the PSG to produce the tone for A above middle C, plug the frequency of 440 Hz into equation 3.

$$TP = 2000000\ Hz\ /\ (16 * 440\ Hz) = 284.09091$$

Because the PSG does not accept fractional values, the tone period for a frequency of 440 Hz is 284.

At this point, you have a value that can be represented as a 12-bit binary number. Now you need to split this value into the coarse and fine tune registers. From a mathematical standpoint, the coarse tune register represents increments of 256 because in the binary numbering system bit 8 represents values of 2^8 or 256. Therefore, the coarse tune register value (CT) is the tone period divided by 256 (to get increments of 256), and the fine tune register value (FT) is the remainder. The equation for this is:

$$TP = 256\ CT + FT$$

From this equation, you can write C statements to calculate the coarse and fine tune register values as follows:

```
CT = TP / 256;
FT = TP % 256;
```

where all variables are of an integer type. In the interest of efficiency, the C statements to use in your programs would look like this:

```
CT = (TP >> 8) & 0x0F;
FT = TP & 0xFF.
```

The first statement shifts the tone period bits to the right by eight bits. This eliminates the fine tune portion of the tone period. The bit-wise AND (&) with 0x0F ensures that only the low four bits of the coarse tune value are used. In the second statement, only the low eight bits are needed; therefore, a bit-wise AND (&) with 0xFF is used.

Continuing with the example above, you have a tone period of 284 to place into the coarse and fine tune registers. The value for the coarse tune register is the integer result of 284 divided by 256, which is 1. The fine tune register holds the remainder of this division here

equal to 28. Therefore, to get a tone of 440 Hz output by the sound chip, the value 28 must be placed in the fine tune register and the value 1 in the coarse tune register. Once you have set the tone period, you are ready to produce the sound.

Noise Period

Register 6 determines the period of the noise. Since this register uses only five of the eight bits available to it, there are 32 different noise frequencies available. Note that because this register stores the period of the wave, a higher value means a lower frequency. For the PSG on the Atari ST, the frequency output ranges from 4 kHz to 125 kHz.

Envelope Generation

What is the envelope? An envelope can be used to control the volume of a voice. The PSG has an envelope generator that does this. There are two attributes to an envelope: shape and period. The envelope *shape control* register, register 13, uses the lower 4 bits to determine the shape of the envelope. Figure 8-3 shows the various values that can be placed in this register and the resulting waveform.

The top of the waveform represents maximum volume and the bottom no volume. For example, the first waveform shown in Figure 8-3 starts at the highest volume and decreases to the lowest volume. The time required for this single iteration of the waveform is called the *envelope period*. The envelope period is set using registers 11 and 12. Register 12 is an eight-bit coarse period value for the envelope, and register 11 is an eight-bit fine period value.

The process of determining the envelope period value is similar to the process used for calculating the tone period. First, you must decide how long you want the envelope to last (measured in seconds). This will be the basic period (ep). The actual envelope period (EP) used in the registers must also be adjusted for the timing rate just as the tone period is adjusted. However, the envelope period uses a timing adjustment value of 2 Mhz divided by 256. Therefore, the equation for calculating the envelope period is this:

EP = (2 Mhz / 256) * ep

For example, if the envelope period is to be 1/2 of a second, the envelope period is as follows:

EP = (2000000 cycles/sec / 256) * 0.5 sec = 3906.25

Since the coarse tune register measures increments of 256, the value for the coarse tune register is 3906 divided by 256, which is 15.

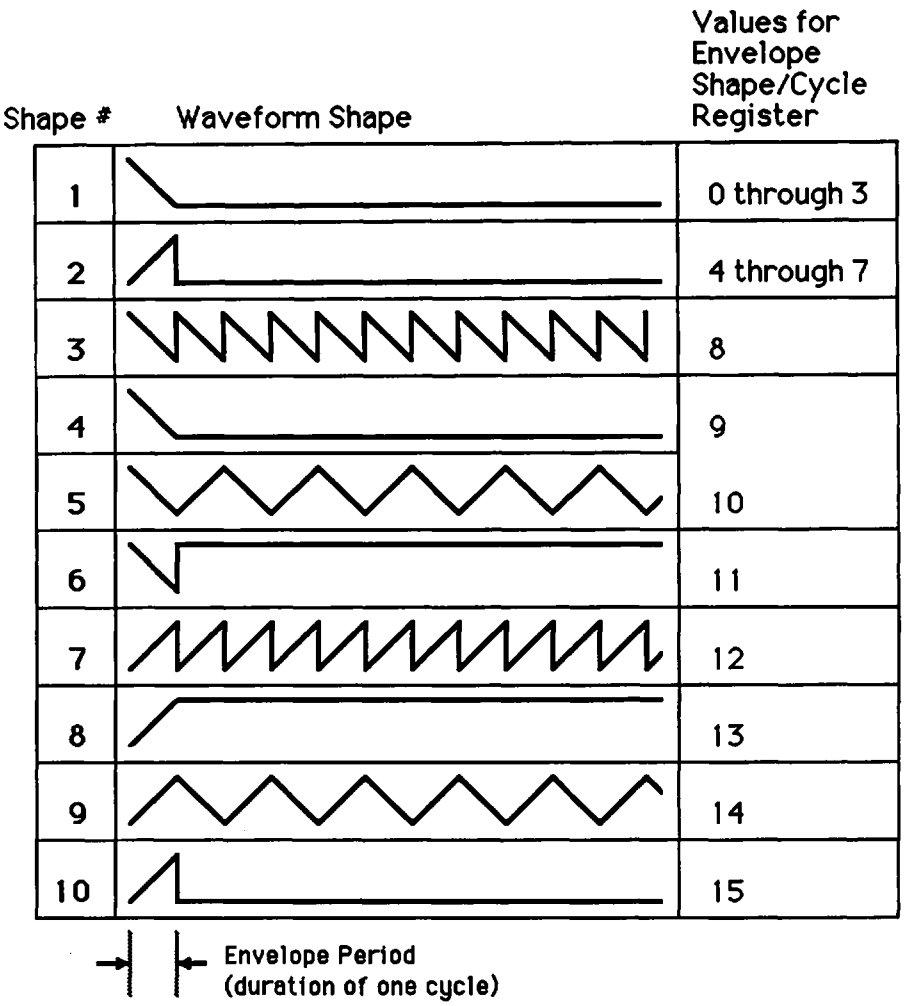


Figure 8-3 Envelope Waveform Shapes

The fine tune register is the remainder of this division, which is 66. Therefore, to get an envelope period of 1/2 of a second, you place 15 into register 12 and 66 into register 11.

Note that in Figure 8-3 waveforms 1, 2, 4, 6, 8, and 10 are only active for one period. After that the volume value is either all on or all off. The other waves are cyclic and keep repeating at one envelope period cycles. The waveforms shown in the figure are the only waveforms available based upon the bit settings in register 13. The actual function of each bit is not important because all combinations are shown here.

Volume Control

Registers 8, 9, and 10 control the volume levels for voices A, B, and C, respectively. The low four bits (bits 0 through 3) control the constant volume level for the associated voice. Because there are four bits, there are 16 different volume levels. The value 15 is the highest volume, and value 0 is the lowest (off). Bit 4 in these registers activates the envelope volume control. If bit 4 is set to 1, the volume for that voice is controlled by the current envelope settings. If bit 4 is 0, the volume is controlled by the value in the volume control register. Therefore, to activate the envelope control, place the value 16 into the volume control register.

Sound Output

To activate the speaker, you must do two steps. First, enable the voice or voices to be output. Then set the volume for the voice or voices. Enabling the voice output is done through register 7, aptly called the voice enable register. This register is designed as an *inverse register*, which means that the bit values have the reverse interpretation to usual. Thus, a bit set to 1 is said to be "off," and a bit set to 0 is said to be "on."

Looking at Figure 8-2, you see that the voice enable register has three parts: the tone enable, the noise enable, and the input/output control. The input/output control is associated with bits 6 and 7. These bits control data flow in to and out of registers 14 and 15. As mentioned earlier, these registers are not used in this book. The tone enable portion of register 7 is located in bits 0, 1, and 2. These bits turn on tone output for voices A, B, and C, respectively. Bits 3, 4, and 5 are the noise enable bits for voice A, B, and C, respectively.

If the tone enable bit is set to 0 (this is an inverse register), the tone specified in the coarse and fine tune register for that voice is output. For example, if bit 0 is set to 0, the tone for voice A is output at the volume indicated in register 8. If bit 0 and bit 2 are both 0, the tones for voice A *and* for voice C are output at their corresponding volume levels.

The noise enable bits work in the same manner. If bit 3 is 0, noise is output on voice A at the indicated volume. The noise used for the voice is specified in register 6. This is the only register used for noise output. Therefore, all voices always have the same noise output at any given time.

Note that any channel or combination of voices can produce both noise and a tone at the same time. For example, if you want to output a tone at 440 hertz, the value 28 must be placed in register 0 and the value 1 in register 1. The register 8 must be set to a nonzero value (15 is the loudest), and register 7 must have the value 62 (binary

111110). This enables voice A to produce the tone. If you put the value 54 (binary 110110) into register 7, noise and tone are output on voice A.

Program SOUNDEMO

The program SOUNDEMO demonstrates the use of the PSG. This is a menu-driven program that allows the user to choose from a variety of sound effects and also allows direct access to the PSG registers so that the user can experiment with different sounds. SOUNDEMO also demonstrates several other features, such as the *alpha mode* of the VDI, which has not yet been covered. The VDI is primarily designed for graphics output. However, since many programs use much text, it would be very inefficient for the program to format text strings and output them as graphic text. The alpha mode of the VDI is used so that the Atari screen can emulate a standard text-based terminal. Alpha mode provides many of the terminal-type functions such as cursor positioning, screen clearing to the end of the line or to the end of the screen, line insert, and delete and cursor positioning.

SOUNDEMO also shows how to access restricted portions of memory. For the program to work properly, it must ensure that the keyclick feature is disabled. This is done by checking one of the reserved system memory locations, disabling the keyclick bit in that value, and replacing the value in the memory location. When the program finishes, the keyclick is restored to its initial state (either on or off). This system variable is located in a restricted portion of memory and must be accessed with a special procedure.

Looking at **main( )**, you can see the control flow of the program (see Listing 8-1). After performing the usual initialization, the program turns off the keyclick by using the **set_keyclick( )** routine. The next step sets the workstation to alpha mode using the **v_enter_cur( )** command. The program then enters a do-loop that controls the display of the menu. First, **v_curhome( )** is called to send the cursor to the home position, which is the upper left corner of the screen. The **v_eeos( )** function clears from the current cursor position to the end of the screen. The menu is then displayed. There are seven sound effects, an option for custom sounds, and a selection that uses the **Dosound( )** function. Finally, there is an exit option. A second do-loop is entered to allow the user to make and verify a selection. If the selection is 0 (exit selection), the program breaks out of the inner loop and goes back to the outer loop. If the selection is not 0, the program calls the appropriate function. If the selection is not valid, the selection is set to −1 (using the default option of the C *switch* com-

Listing 8-1 Program SOUNDEMO

```
/*********************************************************************
     SOUNDEMO.C Sound testing program

     This program demonstrates the uses of the sound chip.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM structures */
#include  <obdefs.h>               /* GEM write modes */

#include  <ctype.h>                /* character macros */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

char      conterm;                 /* console status byte */

/*************************************************
     Application Specific Data
*************************************************/

/* Sound register names */
char *reg_name[] = {
          "Voice A Fine Tune",
          "Voice A Coarse Tune",
          "Voice B Fine Tune",
          "Voice B Coarse Tune",
          "Voice C Fine Tune",
```

Listing 8-1 (continued)

```
        'Voice C Coarse Tune',
        'Noise Period',
        'Mixer Selection',
        'Voice A Volume',
        'Voice B Volume',
        'Voice C Volume',
        'Envelope Period Fine Tune',
        'Envelope Period Coarse Tune',
        'Envelope Shape/Cycle' };

/* Sound chip instructions. To write data into a sound register, use
*    function Giaccess(dat, reg_num | 0x80) where dat is the data
*    to write and reg_num is the register number. The following
*    defines set the write bit for use with Giaccess() (i.e. AFINE
*    puts data into register 0). Note: DO NOT write directly into
*    the mixer register, use function mixer() (see text for the
*    explanation.
*/

#define   AFINE     0 | 0x80
#define   ACOARSE   1 | 0x80
#define   BFINE     2 | 0x80
#define   BCOARSE   3 | 0x80
#define   CFINE     4 | 0x80
#define   CCOARSE   5 | 0x80
#define   NOISEPER  6 | 0x80
#define   MIXER     7 | 0x80
#define   AVOL      8 | 0x80
#define   BVOL      9 | 0x80
#define   CVOL      10 | 0x80
#define   ENVFINE   11 | 0x80
#define   ENVCOARSE 12 | 0x80
#define   ENVCYCLE  13 | 0x80

/* Channel output selection (in octal). Use bitwise AND (&) to
     combine channels, then use the mixer() function. */

#define   TONEA     076
#define   TONEB     075
#define   TONEC     073
#define   NOISEA    067
#define   NOISEB    057
#define   NOISEC    037
#define   ALLOFF    077

char sound_demo[] = {
     /* play note A (440 Hz) for 2 seconds */
     0, 28, 1, 1, 7, 62, 8, 8, 130, 100, 7, 63,

     /* play chord of A, C#, E for 2 seconds */
     0, 28, 1, 1, 2, 194, 3, 1, 4, 123, 5, 1, 7, 56,
     8, 8, 9, 8, 10, 8, 130, 100, 7, 63,
```

Listing 8-1 (continued)

```
     /* "Ramping" sound effect */
     0, 0, 1, 1, 7, 62, 8, 8, 128, 1, 129, 0, 1, 255, 7, 63, 255, 0
};

/*************************************************
     GEM-related Functions
*************************************************/


WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     v_clrwk(new_handle);                /* clear workstation */
     return(new_handle);
}


set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/
```

Listing 8-1 (continued)

```
set_conterm()
/*************************************************
Function: Sets Atari ST global variable location 0x484
Input:    Variable conterm must be set to new value
Output:   None.
Notes:    This function MUST be called using supervisor mode,
          e.g., Supexec(set_conterm)
*************************************************/
{
     *(char *)0x484 = conterm;
     return;
}


get_conterm()
/*************************************************
Function: Gets value of Atari ST global variable location 0x484
Input:    None.
Output:   None. Sets variable conterm.
Notes:    This function MUST be called using supervisor mode,
          e.g., Supexec(get_conterm)
*************************************************/
{
     conterm = *(char *)0x484;
     return;
}


set_keyclick(setting)
int setting;
/*************************************************
Function: Provide access to key click setting.
Input:    setting   = TRUE to turn on
                      FALSE to turn off
Output:   Returns current setting of key click.
Notes:    Uses functions set_cconterm() and get_conterm()
*************************************************/
{
char save_con;

     Supexec(get_conterm);
     save_con = conterm;      /* save current settings */
     if (setting)
          conterm = save_con | 1;       /* key click on */
     else
          conterm = save_con & 0xfe;    /* key click off */
     Supexec(set_conterm);
     return(save_con & 1);         /* bit 0 is key click setting */
}
```

Listing 8-1 (continued)

```
wait(ms)
long ms;
/*************************************************
Function: Provide delay for sound effects routines.
Input:    ms   = delay in milliseconds.
Output:   None.
Notes:    Because the routine is implemented in C. The delay
     is only approximate. The inner loop takes 1 ms.
*************************************************/
{
long i;

     for(; ms > 0; ms--)
          for (i = 125; i > 0; i--)
               ;
     return;
}

mixer(ch)
int ch;
/*************************************************
Function: Sets mixer register in sound chip.
Input:    ch   = integer value to put in register.
Output:   None. Register 7 set.
*************************************************/
{
register temp;

     temp = Giaccess(0,7);          /* read current value */
     temp = temp | 077;             /* turn off current mixer setting */
     ch = ch | 0300;                /* set bits 6,7 to prevent */
                                    /* changing of I/O bits */
     Giaccess(temp & ch, MIXER);    /* write new setting */
     return;
}

clear_sound()
/*************************************************
Function: Sets sound registers to 0.
Input:    None.
Output:   None. Turns off all sound generation.
*************************************************/
{
register  i;

     for(i = 0; i < 7; i++)
          Giaccess(0, i | 0x80);
     mixer(ALLOFF);                 /* handle reg 7 separately */
     Giaccess(0, AVOL);
     Giaccess(0, BVOL);
     Giaccess(0, CVOL);
     return;
}
```

Listing 8-1 (continued)

```
enter_notes()
/*************************************************
Function: Allows user to enter sound registers from the keyboard.
Input:    None.
Output:   None.
*************************************************/
{
int  reg,                          /* input register */
     dat;                          /* register data */

     clear_sound();
for(;;)                            /* loops until neg reg entered */
{
     v_curhome(screen_vhandle);
     v_eeos(screen_vhandle);
     printf("Current values:\n");
     for(reg = 0; reg <= 13; reg++)      /* display registers */
          printf("%2d> %-30s %4o\n",
               reg, reg_name[reg], (Giaccess(0, reg) & 0377));
     printf("\n");

     printf("Enter register number to change\n");
     printf("or a negative value to end: ");
     scanf("%d", &reg);
     if (reg < 0)                        /* neg reg ends function */
     {
          clear_sound();
          return;
     }
     printf("Enter data to store: ");
     scanf("%o", &dat);                  /* get data */

     if (reg == 7)                       /* if mixer, user mixer() */
          mixer(dat);
     else if (reg <= 13)                 /* else write data */
          Giaccess(dat, reg | 0x80);
     /* ignore all else */
}
}

siren()
/*************************************************
Function: Sound effect for European siren.
Input:    None.
Output:   None.
*************************************************/
{
     printf("\nPress any key to stop:\n");
     clear_sound();
     mixer(TONEA);                 /* set tone output on channel A */
     Giaccess(15, AVOL);           /* set max volume on A */
     do
     {
```

Listing 8-1 (continued)

```
            Giaccess(254, AFINE);     /* set higher tone on A */
            Giaccess(0, ACOARSE);
            wait(350L);               /* wait 350 ms */
            Giaccess(86, AFINE);      /* set lower tone on A */
            Giaccess(1, ACOARSE);
            wait(350L);               /* wait 350 ms */
        } while (!Cconis());          /* check if any key pressed */
        clear_sound();                /* turn off sound */
        Crawcin();                    /* capture key pressed */
        return;
}

gunshot()
/*************************************************
Function: Sound effect for gunshot.
Input:    None.
Output:   None.
*************************************************/
{
        clear_sound();
        Giaccess(15, NOISEPER);             /* set max noise period */
        mixer(NOISEA & NOISEB & NOISEC);    /* set noise on all channels */
        Giaccess(16, AVOL);                 /* set volume to be */
        Giaccess(16, BVOL);                 /* controlled by envelope */
        Giaccess(16, CVOL);                 /* generator */
        Giaccess(16, ENVCOARSE);            /* set envelope period */
        Giaccess(0, ENVCYCLE);              /* set cycle type */
        return;
}

explosion()
/*************************************************
Function: Sound effect for explosion.
Input:    None.
Output:   None.
*************************************************/
{
        clear_sound();
        Giaccess(8, NOISEPER);              /* set noise period */
        mixer(NOISEA & NOISEB & NOISEC);    /* noise on all channels */
        Giaccess(16, AVOL);                 /* envelope control on */
        Giaccess(16, BVOL);                 /* all channels */
        Giaccess(16, CVOL);
        Giaccess(56, ENVCOARSE);            /* envelope period */
        Giaccess(0, ENVCYCLE);              /* cycle type */
        return;
}

laser()
/*************************************************
Function: Sound effect for laser.
Input:    None.
Output:   None.
*************************************************/
```

Listing 8-1 (continued)

```
{
register sweep;

     printf('\nPress any key to stop.\n');
     clear_sound();
     mixer(TONEA);                        /* use channel A tone only */
     Giaccess(15, AVOL);                  /* set max volume */
     do
     {
          for (sweep = 48; sweep <= 70; sweep++)
          {                               /* decreasing tone sweep */
               Giaccess(sweep, AFINE);    /* on channel A */
               wait(3L);                  /* 3 ms delay */
          }
     } while (!Cconis());                 /* check for keypress */
     Giaccess(0, AVOL);                   /* turn off sound */
     Crawcin();                           /* capture key */
     return;
}


bomb()
/***************************************************
Function: Sound effect for bomb.
Input:    None.
Output:   None.
***************************************************/
{
register sweep;

     clear_sound();
     mixer(TONEA);                        /* tone on channel A */
     Giaccess(15, AVOL);                  /* max volume */
     for(sweep = 48; sweep <= 160; sweep++)
     {                                    /* decreasing tone sweep */
          Giaccess(sweep, AFINE);
          wait(25L);
     }
     explosion();                         /* end effect */
     return;
}


whistle()
/***************************************************
Function: Sound effect for whistle.
Input:    None.
Output:   None.
***************************************************/
{
register sweep;
```

Listing 8-1 (continued)

```
    clear_sound();
    Giaccess(1, NOISEPER);          /* set noise period */
    mixer(TONEA & NOISEB);          /* set channel output */
    Giaccess(15, AVOL);             /* max vol on A */
    Giaccess(9, BVOL);              /* lesser vol on B */
    for(sweep = 64; sweep >= 32; sweep--)
    {                               /* increasing tone sweep */
        Giaccess(sweep, AFINE);
        wait(20L);
    }
    Giaccess(0, AVOL);              /* stop tone sound */
    wait(150L);                     /* wait */
    Giaccess(15, AVOL);             /* restart tone sound */
    for(sweep = 80; sweep >= 48; sweep--)
    {                               /* increasing tone sweep */
        Giaccess(sweep, AFINE);
        wait(35L);
    }
    for(sweep = 48; sweep <= 104; sweep++)
    {                               /* decreasing tone sweep */
        Giaccess(sweep, AFINE);
        wait(15L);
    }
    clear_sound();                  /* end sound */
    return;
}

race_car()
/*************************************************
Function: Sound effect for race car.
Input:    None.
Output:   None.
*************************************************/
{
register  sweepf,                   /* fine register sweep */
          sweepc;                   /* coarse register sweep */

    clear_sound();
    Giaccess(6, BCOARSE);           /* set B tone value */
    Giaccess(15, BFINE);
    mixer(TONEA & TONEB);           /* tones on A and B */
    Giaccess(15, AVOL);             /* max volume on A */
    Giaccess(10, BVOL);             /* lower volume on B */
    for(sweepc = 10; sweepc >= 5; sweepc--)
    {                               /* increasing tone on A */
        Giaccess(sweepc, ACOARSE);
        for(sweepf = 255; sweepf >= 0; sweepf--)
        {
            Giaccess(sweepf, AFINE);
            wait(2L);
        }
```

Listing 8-1 (continued)

```
     }
     for(sweepc = 8; sweepc >= 3; sweepc--)
     {                              /* increasing on A at higher tone */
          Giaccess(sweepc, ACOARSE);
          for(sweepf = 255; sweepf >= 0; sweepf--)
          {
               Giaccess(sweepf, AFINE);
               wait(4L);
          }
     }
     for(sweepc = 6; sweepc >= 2; sweepc--)
     {                              /* increasing on A higher still */
          Giaccess(sweepc, ACOARSE);
          for(sweepf = 255; sweepf >= 0; sweepf--)
          {
               Giaccess(sweepf, AFINE);
               wait(6L);
          }
     }
     clear_sound();                 /* end sound effect */
     return;
}

/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */
int  select;                        /* menu selection */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

int  key_click;                     /* save key click setting */

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }
```

Listing 8-1 (continued)

```
     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */

/*************************************************
Application Specific Routines
*************************************************/

/* turn off key click */
     key_click = set_keyclick(FALSE);

     v_enter_cur(screen_vhandle);  /* use alpha mode */
     do
     {
          v_curhome(screen_vhandle);
          v_eeos(screen_vhandle);  /* clear screen */
          printf("\n\n\n\nSound Demonstration Program\n\n\n");
          printf("  1> European Siren\n");
          printf("  2> Gunshot\n");
          printf("  3> Explosion\n");
          printf("  4> Laser\n");
          printf("  5> Falling Bomb\n");
          printf("  6> Whistle\n");
          printf("  7> Race Car\n");
          printf("  8> Enter Your Own\n");
          printf("  9> Dosound() Demo\n");
          printf("  0> Exit Program\n\n");

          do
          {
               printf("Enter your selection: ");
               scanf("%d", &select);

               if (select == 0)
                    break;              /* exit loop */

               switch (select)
               {
               case 1:   siren();
                         break;
               case 2:   gunshot();
                         break;
               case 3:   explosion();
                         break;
               case 4:   laser();
                         break;
               case 5:   bomb();
                         break;
               case 6:   whistle();
                         break;
```

Listing 8-1 (continued)

```
                case 7:   race_car();
                          break;
                case 8:   enter_notes();
                          break;
                case 9:   Dosound(sound_demo);
                          break;
                default:  select = -1;
                          break;
                }
            } while (select < 0);
        } while (select != 0);              /* end do loop */

/*************************************************
Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     set_keyclick(key_click);       /* restore key click */
     v_exit_cur(screen_vhandle);    /* exit alpha mode */
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
```

mand) and the loop is repeated requesting another selection. When the sound functions return, the inner loop is exited and the outer loop displays the menu again. When the exit selection is requested, the outer loop exits, the keyclick state is restored, and **v_exit_cur( )** is called to exit the alpha text mode of the workstation. Then the workstation is closed and the program is finished.

Protected Memory Access

In the application functions section of the program, look at the **set_keyclick( )** routine. Its one parameter determines whether the keyclick is turned on or off. If the parameter is TRUE, the keyclick is turned on. If it is FALSE, the keyclick is turned off. The **set_keyclick( )** function returns the setting of the keyclick before it is changed. The first statement in **set_keyclick( )** is a function that provides the special access mentioned above. This is needed because the current setting of the keyclick is located in a protected section of memory. A standard program does not have access to this location in memory so an error would occur if the program tried to read it. To gain access to this memory, the program needs to set the microprocessor into what is called *supervisor* mode. This allows the program to access virtually any location in memory. The **Supexec( )** function is one of two functions that can be used to put the processor into supervisor

mode. **Supexec( )** is an extended BIOS function that places the processor into that mode and executes the code starting at the address specified by its parameter. In **set_keyclick( )**, the parameter is the address of the function **get_conterm( )**. When **Supexec( )** is called, the processor is set to supervisor mode and **get_conterm( )** is executed. When **get_conterm( )** is done, the processor is reset to user mode and **set_keyclick( )** continues. Another function can be used to set the processor to supervisor mode called **Super( )**. This is a GEMDOS function that also has one parameter: the address for the stack location. Once **Super( )** is called, the processor stays in supervisor mode until another call to **Super( )** is made. The difference between the two functions is that **Supexec( )** does not allow BIOS or GEMDOS calls. It is simply a quick means to allow access to protected locations without worrying about stacks and modes.

Function **get_conterm( )** simply reads memory location hexadecimal 0x484 into the global variable conterm. Location 0x484 contains the console settings. Appendix D shows a set of addresses that are "cast in stone" in the memory. Specified by the Atari ST designers these locations are guaranteed not to change through future revisions to the ST. Your programs can obtain and change the information regarding the ST itself by reading and writing to these locations in supervisor mode. Many of these values are better left unchanged, however.

In **set_keyclick( )**, once the current console settings are retrieved, the keyclick setting is saved in the variable **save_con**. If the requested new setting is TRUE, the program wants the keyclick setting turned on. The previous console settings are bit-wise ORed with the value 1 because bit 0 contains the keyclick setting. By using the bit-wise OR with the value 1, bit 0 is set to 1 regardless of its previous value. On the other hand, if the keyclick is to be turned off, the current value in **save_con** is bit-wise ANDed with hexadecimal FE. This sets bit 0 to 0 and leaves all other bits unchanged. After the new console setting has been determined, **Supexec( )** is called again to set the new setting. Function **set_conterm( )** stores the value in variable **conterm** at the appropriate location. When **set_conterm( )** is finished, control returns to **set_keyclick( )**, which returns the previous setting of the keyclick.

The reason the keyclick must be turned off for this program is that the keyclick feature also uses the sound chip to produce the clicking sounds. This would interfere with the use of the chip by the program.

Now look at function **wait( )**. This function simply provides a delay when it is called. The loop in the function provides an approximate one millisecond (1/1000 second) delay. The parameter determines the approximate number of milliseconds. **Wait( )** is used in several locations to produce some of the sound effects.

PSG Access

Access to the sound chip is provided through the XBIOS function **Giaccess( )**, which has two parameters. The first is the value to be put in the register; the second is the register to use. If the register parameter has a value between 0 and 15, **Giaccess( )** performs a read of that register and returns the current value. If the register number is bit-wise ORed with the value 0x80 hexadecimal (that is, has its high bit set), **Giaccess( )** writes the value of the first parameter into the register. In the application-specific data section of SOUNDEMO, there is a set of defined constants that list the 14 registers to be used (AFINE, ACOARSE, BFINE, BCOARSE, and so on). These constants specify the writing mode for function **Giaccess( )** and are used to provide more readable code.

The next set of defined constants, TONEA, TONEB, TONEC, NOISEA, NOISEB, NOISEC, and ALLOFF, are the bit setting for register 7, the voice enable register. Note that these constants are octal values. Use the bit-wise AND to create any combination of tone and noise on any of the voices. For example, if you want tone on voice A and noise on voice C, use the value TONEA & NOISEC. Remember that register 7 is an inverse register so the bit-wise AND is used instead of the bit-wise OR.

Register 7 controls the mix of sounds produced by the sound chip. This register also controls the input and output through registers 14 and 15. When setting the bits in the voice enable register, also called the mixer register, you do not want to alter the settings of bits 6 and 7. Therefore, when changing the voice enable bits, you must first read the current setting in register 7, change the voice enable bits while retaining the settings for bits 6 and 7, and place the new value into register 7. If you do not do this, you may alter the data flow through the PSG and cause the system to stop working.

To facilitate the protection of bits 6 and 7 in the mixer register, function **mixer( )** is provided in SOUNDEMO. This function has one parameter, which is the voice selection value to be placed in the mixer register. Function **mixer( )** first reads the current value in register 7 into variable temp. All voices are turned off by setting the low six bits to 1 through the bit-wise OR with octal value 077. Next the parameter **ch** has its high bits set to 1. Then the variables **temp** and **ch** are bit-wise ANDed together. Variable temp holds the current bit settings for the I/O ports and **ch** holds the bit settings for the voice **mix**. When ANDed together, the result is a single value with the proper I/O port and mixer bit settings. For example, **temp** is set to the binary value ab111111, where a and b are the bit settings for bits 6 and 7. The variable **ch** is set to the binary value 11uvwxyz, where the letters

stand for the voice mix bit settings. When ANDed together, you get

```
  ab111111
& 11uvwxyz
----------
  abuvwxyz
```

so that the port settings are not changed and the mixer settings are set to their new values.

The next function in SOUNDEMO, **clear_sound( )**, simply clears the sound registers so that all sound generation is stopped. The function goes through registers 0 through 6 and sets them to 0. It sets the mixer register to ALLOFF. The volume for each channel is then set to 0. Technically, to turn off a channel, its volume must be set to 0. According to the PSG documentation, simply disabling the voice using the mixer register does not actually turn that channel off. Although no sound may be coming out, the channel is still active and the PSG is still processing it.

Using the PSG

Function **enter_notes( )** corresponds to selection number 8 on the menu, entitled "Enter Your Own." This function clears the screen and displays the current values for registers 0 through 13. The user is then allowed to enter a register number and a value to place in that register. If a negative register number is requested, the function exits.

The first step in **enter_notes( )** is to clear the sound registers using the **clear_sound( )** function. Next an infinite loop is entered, which has the same purpose as the outer control loop in function **main( )**. Within this loop, the function clears the screen, displays the current values, and requests a register number. If the register number is negative, the sound registers are cleared again and the function returns to the calling procedure. If the register number is not negative, data is requested. Once the data is entered, the function checks if the register number requested is register 7. If so, function **mixer( )** is used to store the data. Otherwise, **Giaccess( )** is used to store the new value. If a register number greater than 13 is entered, the data is ignored. The infinite loop keeps repeating this procedure until a negative register number is entered. The current sound register values are listed in octal because octal is easier to use in this case, particularly for register 7. The **scanf( )** function requests the register number as a decimal value and the data as an octal value.

The Sound Stage

The next set of functions is the sound effect functions listed in the menu. The first function is function **siren( )**. This function emulates the European emergency vehicle siren, which is a high tone followed by a low tone. This sequence repeats over and over. The first step for **siren( )** is to clear the sound registers using **clear_sound( )**. Only one voice, voice A, will be used for this effect so that the mixer is set to output a tone on voice A and the voice is set for maximum volume. A do-loop is used to produce the repeating sequence of tones. First the tone period of the higher note is set and the function waits for 350 milliseconds (about 1/3 of a second). While the program is waiting, the sound chip is producing the specified sound. After the waiting period, the lower tone is set and another 350-millisecond wait occurs. The loop then repeats until a key is pressed. When a key is pressed, the sound chip is cleared and the value of the key press is captured using the **Crawcin( )** call. The function then returns to the calling procedure. It is important to capture the key press; otherwise it remains in the keyboard buffer and appears on the selection line for the main menu.

Since the PSG is constantly processing the current register settings, the sound chip works independently of your program. Once the sound registers have been set, that sound is continually produced until the sound register is changed or the volume for the voices is turned off. For example, in **siren( )** the tone continues for 350 milliseconds while the program is in the **wait( )** function. The tone changes by resetting the tone period values. All other registers are unaffected. In the case of an envelope setting, as soon as a new envelope shape is selected that envelope begins to cycle. You may change tone periods, noise periods, volumes, envelope periods, and envelope shapes at any time. Any change you make affects only that register. This relieves your program of having to reset the sound registers for each new sound you want to create.

Function **gunshot( )** provides a reasonable facsimile of a gunshot sound effect. This function utilizes the envelope available on the sound chip. First, **gunshot( )** uses **clear_sound( )** to clear the sound chip. Then the noise period is set to 15 and the mixer is set to output noise on all three voices. To allow the envelope shape to control the volume, the volume registers for the three voices are set to the value 16. Next the envelope coarse period value is set and the envelope shape is chosen. As soon as the envelope shape is chosen, the PSG begins cycling through the envelope (in this case there is only one cycle). Thus a gunshot effect is heard.

There are two reasons why all voices are used to output noise for

the gunshot. First, the more voices used, the louder the sound becomes. Second, the noise is generated by very short random tones. If you separate the noise from each voice, the noise from each voice may sound alike but it is not exactly the same. This gives the gunshot its piercing texture.

Function **explosion( )** is very similar to **gunshot( )**. The noise has a period of 8 instead of 15, which means that the frequency is higher. The coarse envelope period for **explosion( )** is longer, so the sound is of longer duration.

Function **laser( )** uses a *sweep effect*. This is a constantly increasing or decreasing tone, which causes a continuous changing, or "sweeping," effect. Function **laser( )** uses voice A at its maximum volume. The fine tune register for that voice A is swept from a value of 48 to a value of 70 with a three-millisecond delay between each change. This gives quick bursts of decreasing tone chirps. Laser continues until a key is pressed. The function ends by turning off voice A.

Function **bomb( )** uses a combination of sound effects. First is a long sweep of a decreasing-frequency sound (the whistling sound of a falling bomb) that ends with the sound of an explosion created by function **explosion( )**.

The **whistle( )** function produces the "wolf" whistle that a macho man makes toward a pretty female. Function **whistle( )** uses a combination of tone and noise since a real whistle has a slight rushing air sound. The tone on channel A is set to the maximum volume. The noise on channel B is set at a lower volume. Then the function goes through two increasing-frequency tone sweeps and then a decreasing-frequency tone sweep.

The **race_car( )** function emulates the sound of a race car that is accelerating and shifting gears. It uses two different tones on two different channels, A and B. Channel B contains the low, grumbling sound and channel A contains the increasing-frequency sound. Channel A has a higher volume than channel B. The function goes through three different sweeps representing the three different gears. Each loop starts channel A at a higher frequency than the previous loop.

The Dosound( ) *Function*

The last function to be discussed is an XBIOS function called **Dosound( )**. It is called in case 9 of the **switch** statement. Function **Dosound( )** provides and processes a sequence of sound operations that run concurrently with your program. Thus, while **Dosound( )** is generating a sequence of sounds, your program can do something else.

The parameter for **Dosound( )** is a pointer to a sequence of *bytes* that represent the settings of the PSG registers and time delays to be

executed between each byte in the list. Under the application-specific data in the program the last variable declared is the character array **sound_demo**. Function **Dosound( )** uses pairs of bytes to set the registers of the PSG. The first byte is a command and the second a data value (see Table 8-1). Command numbers from 0 through 15 are register commands. These take the next value in the list as the data to be stored in that register. For example, the first two elements in the **sound_demo** array are 0 and 28. This places the value 28 into register 0. The next two values, 1 and 62, indicate that the value 62 is to be placed in register 1. Then the value 62 is placed in register 7 and the value 8 into register 8. As calculated earlier, the value 28 in the fine register and 1 in the coarse register corresponds to a tone at 440 Hz. Placing 62 into register 7 turns on tone generation for voice A, and placing 8 into register 8 turns the volume on half way.

Table 8-1: Dosound( ) **Process Commands**

Command Number	*Function*
0 through 15	Place next byte into register.
128	Place next byte into temporary register.
129	Use next 3 bytes for sweep effect. Byte 1: register to use. Byte 2: increment value. Byte 3: termination value. Take value in temporary register and put it into register specified. Increment register by increment value until it equals the termination value.
130 through 254	Use next byte for timing delay measured in 1/50 second.
255	If next byte is 0, terminate the Dosound( ) process. Otherwise, use value for timing delay.

Commands 130 through 255 take the next value in the list as the argument and use this value as a wait period before moving to the next command. The value specifies the number of cycles to wait based upon a 50-Hz clock rate. Therefore, an argument of 50 causes a delay of one second, an argument of 25 causes a delay of 1/2 second, and so on. Commands 128 and 129 are used together to produce a sweep effect. Command 128 takes the next value in the list and places it in a temporary register; the documentation does not state where this register is located. Command 129 uses the next *three* values as arguments to the function. The first argument is the register number to use, the second argument is used as the increment value, and the

third argument is the termination value. Command 129 takes the value in the temporary register (set by command 128) and places it into the register specified by the first argument. This register is incremented by the increment value of the second argument. The sweep stops when the register value *equals* the termination value of the third argument.

The contents of the **sound_demo** array play the following sequence of sounds: first the note A is played for two seconds, then a chord is played for two seconds, and a sweep is produced. Toward the end of the first line in the **sound_demo** array, command 130 is followed by 100. This causes the note A (440 Hz) to be played for two seconds. The last command on the first line, 7 with a data value of 63, turns the sound off. The channel is not turned off, just the sound output. The next set of commands plays a chord. Note A is stored in the voice A tone period registers, the note C-sharp is stored into the voice B tone period registers, and note E is stored into the tone period registers for voice C. The value 56 is placed into register 7, which turns on tone output for all three voices. The value 8 is used as the volume for each voice and is placed into registers 8, 9, and 10. The command 130 is again followed by the value 100 to create a two second sound. Then sound generation is turned off.

The last section produces the sweeping, or "ramping," effect to demonstrate commands 128 and 129. First a 0 is put into register 0 and 1 is put into register 1. Then voice A is enabled, and the volume is set. Command 128 places the value 1 into the temporary register. Command 129 has three arguments: 0, 1, and 255. When command 129 begins, the value 1 (from the temporary register) is placed into register 0. This register is incremented from 1 to 255. At this point, **Dosound( )** continues with the next instruction in the list. The next command, 7 with the data 63, disables the voices. The last command, 255 0, is the termination command to tell **Dosound( )** that no more commands are to be processed. This causes the sound output to stop and the end of the **Dosound( )** function.

When you run the SOUNDEMO program, you may notice that when you execute the falling bomb, whistle, and race car sounds, the program control does not return to the menu until the sound effect has been completed. With the **Dosound( )** demo, **Dosound( )** plays its note, chord, and the sweeping effect but the menu is redisplayed while these sounds are being output. Thus, with the **Dosound( )** function, you can have your program playing a song while it is calculating or displaying something on the screen. The **Dosound( )** function is effective only if you want to play a sequence of tones and noises. For any type of sound effect, especially an effect that requires control over the sweep time, you must write a function for the sound effect that uses the **Giaccess( )** function.

Play around with SOUNDEMO by entering your own values into the sound chip. You should also try to make your own sound effects. It can be fun!

The other special option that might be of general interest to you is MIDI capability. MIDI stands for Musical Instrument Digital Interface. The use of this port requires an understanding of the MIDI protocol and hardware interfacing. Because of the length of this topic, it is not covered here. However, if you are interested in communicating with MIDI, you should obtain the MIDI specifications document. Check with your local music store about this.

The remainder of this book discusses the use of the AES. The AES provides your program with access to menus, windows, the mouse, and dialog boxes.

CHAPTER NINE

Application Environment Services: The AES

Until now this book has covered one of the major components of GEM called the Virtual Device Interface (VDI). The VDI is a device-independent method for producing graphic output from a program to any device. In this chapter, another important section of GEM is introduced—the Application Environment Services (AES). You can see how the AES works, what it contains, and how it is used.

Introduction to the AES

By now, you should have encountered many of the features provided by GEM such as windows, icons, and dialog boxes. The AES is the portion of GEM that allows your program, the application, to interact with the Graphics Environment Manager (GEM). The AES is designed to handle all user interaction with the desktop and desktop objects in selecting an item, resizing or moving a window, moving the sliders of a window, or choosing a new window. All these operations are handled by the AES, and the result of these user actions is reported to the current application program.

AES Components

In Chapter 1, you saw that the AES consists of five main pieces: the menu/alert buffer, desk accessory buffer, the shell, the limited multitasking kernel and dispatcher, and subroutine libraries. The menu/alert buffer is simply an area in memory that the AES uses

when handling menus and alert boxes. When a menu or alert box is placed on the screen, it may cover some object on the screen. To keep the screen neat and tidy, the image under the menu or alert box must be replaced when the user has finished with the menu or alert box. This gives the illusion that the desktop is a three-dimensional surface. However, since the screen is a two-dimensional bit map, once the menu or alert box is written on this bit map the previous contents are lost. Therefore, to maintain the illusion, the AES saves the portion of the screen that is covered in the menu/alert buffer. Then when the menu or alert box is no longer needed, the content of the buffer is copied back to the screen bit map, which restores the original image. Since the size of this buffer is one-fourth the size of the screen memory alert boxes and menus may not be bigger than this.

The desk accessory buffer is a portion of memory that contains the accessory programs listed under the DESK menu. When the system is *booted* (turned on or reset), any files on the disk with a file type of ".ACC" are considered to be desk accessories. These files are loaded into the desk accessory buffer and can be accessed through the Desk menu.

The shell is the segment of the AES that handles the execution and termination of application programs. Any time a program is initiated, the shell takes care of the system initialization to start running the program. When the program is finished, the shell handles any clean-up and termination procedures.

The limited multitasking kernel and dispatcher consists of three different parts: the desk accessories, the screen manager, and the dispatcher. *Multitasking* simply means that the system can run more than one task at a time. GEM is a multitasking system. The dispatcher divides the CPU time among the currently executing tasks: the application, the desk accessories, and the screen manager. The dispatcher ensures that no one task hogs the central processor and that all tasks are able to be processed. The screen manager handles user interaction with the desktop. Desktop areas include the border portion of an active window, menu selection, and any portion of the desktop not inside the current active window (such as icons or selecting another window). After such an interaction is completed, the screen manager informs the currently running application of the result of the user actions. The currently running application is the application that owns the currently active window (the topmost window). The last part of the kernel is the desk accessories. A desk accessory is an application that supplies some handy function for the user and usually performs just one function. Accessories generally don't use the menu bar and require only a small portion of the screen, while full applications use the entire screen. The control panel is an example of a desk accessory. GEM loads up to six accessories when

it is booted. The dispatcher allows only three accessories to be active (running) at any one time. The actual number of desk accessories loaded into your system depends on the amount of memory you have and the size of the accessories themselves.

AES Definitions

One of the great benefits of using GEM is that it provides a standard interface for all programs. Users can quickly learn a new program because most of the operational functions such as selecting menus, moving the cursor with the mouse, and using windows are the same for all programs. Programmers don't have to worry about interfacing with the user because this is done by GEM. To fully utilize the GEM user interface, you need to understand the terminology and concepts used in the rest of this book. The section below provides you with the definitions you need.

Menus

At the top of the screen, there is a line of text, called the *menu bar* which provides the user with a set of operations the program can perform. For example, the desktop has a menu bar that lists the items Desk, File, View, and Options. These names are called *menu titles.* Each title refers to a set of menu selections that perform a particular function. Under the Desk title, the first item is an information function that provides information to the user about the program currently running. The second item is a dashed line. The remaining items are the desk accessories available to the user. The File menu provides the user with file activities such as creating, opening, saving, and possibly deleting files on the disk. The File menu also contains the Quit option so the user can exit the program. The format for the Desk and File menus should always follow the format listed here. This enables the user to find these basic options. Do not worry about the names of the desk accessories to include in the Desk menu. GEM handles this aspect of the menu when your program is executed. Any other menu titles on the menu bar are application-dependent. You may create as many menus as can fit across the screen. Just make sure that the titles accurately describe the functions contained in the menu.

The entries in the menu are called *menu items.* Menu items have a particular format that you should follow. The menu item is a brief one- or two-word command indicating the function to be performed. Optionally, there may be a character to the left of the menu item description, which represents the key on the keyboard that performs the same function. Usually this refers to a control code where the user

presses the Control key and a letter key at the same time. For example, your program might have a Q next to the Quit menu item, meaning that the user can press Control-Q to exit the program instead of using the mouse to select the Quit option.

Menu items can have a number of attributes. When a menu item is shown in its normal state, it is said to be enabled. When it is displayed at only half intensity, it is disabled. The AES does not allow you to select a disabled item. A menu item may be checked. When an item is checked, there is a check mark in the leftmost character position next to that item. A checked item usually indicates an active toggle. For instance, a text editor might use a checked item for text justification. When justification is on, the check mark is displayed. When it is off, the check is not displayed. If this item is selected, the check mark toggles to its opposite state, that is, if it is on, it goes off and vice-versa.

Boxes

GEM uses three types of boxes: an error box, a dialog box, and an alert box. An error box is used by a program when a system error has occurred and the program can't perform its operation. The error box has a predefined format that says an error has occurred and provides the number of the error.

A dialog box is a rectangular box on the screen that requests information from the user. The information requested can be a file name or a complex data entry screen as might be used in an inventory program. The actual contents of a dialog box is determined by the programmer. An example of a dialog box occurs in the Options menu of the desktop. The Set Preferences selection causes a dialog box to be displayed so that the desktop options may be set.

An alert box is a special form of dialog box. It advises the user of some pertinent condition and allows the user to verify a particular operation. For example, if you are editing a program and try to read in a file that is too large for the editor to handle, an alert box appears to indicate the problem. The box may have options to choose to alleviate the problem.

Windows

A *window* is something that must be very familiar by now. It is an area of the desktop that provides communication between the application and the user. The window consists of several components. These are the title bar, the information line, close box, full box, move bar, size box, up arrow, down arrow, left arrow, right arrow, vertical scroll bar and vertical slider, horizontal scroll bar and horizontal

slider, and the work area. When you are at the desktop, the directories for your disks are held in windows. You can look at these for examples of the window components. The title bar is the area of the window that holds the directory currently being shown. This will usually be something like "A:\" or "A:\" followed by a subdirectory name. The information line is the area of the window that shows the number of bytes used and the number of items displayed in that directory. Both the title bar and the information line can hold up to 80 text characters. The close box is located at the left end of the title bar and contains the small black box with the white X on it. When the user clicks the mouse inside the close box the AES tells the application that the user has requested that the window be closed. The full box is at the right side of the title bar. It has a white diamond on a black background. When the window is at its "normal" size and the full box is clicked, the window expands to its maximum size as specified by the application. If the full box is clicked again the window returns to its original size. The move bar occupies the same area as the title bar. When the user presses and holds the mouse button on the move bar, the AES provides a half-intensity outline of the window and allows the user to drag this outline to any location on the screen. When the button is released, the AES tells the application of the new requested window location. The size box is located in the lower right corner of the window and has a white slash on a black square. When the mouse button is pressed on the size box, the AES displays a half-intensity outline of the window and allows the user to change the size and shape of the window by dragging the outline. When the button is released, the AES indicates to the application the new requested size of the window.

The up arrow and down arrow are located on the right border of the window and are indicated by corresponding up and down arrow figures. When either arrow is clicked, the AES informs the application of this event. The application redraws the window as appropriate. For example, consider the desktop display of your directories in the text mode (set in the View menu). When you press the up arrow, the listing moves up by one line. The down arrow moves the listing down one line. In the icon mode, the up and down arrows move the icons up and down one row of icons. Since one row of icons is taller than one row of text, the application must consider what is being displayed. The left and right arrows work analogously and move the window to the left and right over the contents. The vertical scroll bar and slider are in the area located between the up and down arrows. The slider represents the relative location in the entire file of the contents currently shown in the window and the relative size of the window to the total size of the data. For instance, if the slider is at the bottom of the scroll bar, the information in the window is the very end of the

data. If the size of the slider is two-thirds of the total length of the scroll bar, two-thirds of the total amount of data is being displayed in the window. When the user clicks the colored area of the scroll bar, the window moves one page either up or down depending on whether the user clicks the area above or below the slider, respectively. When the user presses the mouse button while located on the slider itself, the user can drag the slider anywhere within the scroll bar. When the button is released, the appropriate section data is displayed. The horizontal scroll bar and slider work in the same way as the vertical scroll bar and slider.

The work area is the area of the window where the data is displayed. It is essentially the part of the window not covered by any of the components listed above. The work area is the only required portion of the window. All other components are optional. When a window is created, the application tells the AES what components are to be included in the window. Whenever the AES draws the window, it draws only those components that have been included in the window.

Messages and Events

All user interactions with the border components of the window are handled by the AES. When the user interaction is complete, the AES reports the results to the application using a mechanism called a *message pipe*. A message pipe is simply a means for communication between one program and another. For example, when the user requests a window closed by clicking the close box, the screen manager sends an appropriate message to the application owning that window. When a message is sent to an application, it is called an *event*. An event controls the flow of the application programs. There are keyboard events, mouse events, mouse button events, message events, and timer events. Any one of these can occur at any time. As you can see, the application essentially just sits around, waits for one of these events to occur, and then acts upon it. For example, an application such as the desktop waits for an event. When it receives the message that the window is to be closed, the screen manager sends a message to the desktop. The desktop receives the event, determines what type of event it is, gets the data regarding the event, and performs the appropriate action—namely, closing the window.

Libraries

The AES provides routines to handle all the objects described above: menus, dialog boxes, alert boxes, windows, events, and so on. These routines are held in *libraries* that are included as part of the AES in memory. They are accessible to all applications. Since each applica-

tion does not need its own copy of these routines, memory is conserved and the applications require less space on the disk.

The routines are divided into library groups such as an event library, an application library, a menu library, an object library, a form library, a graphics library, and a scrap library. Other libraries are file selector, window, resource, and shell. The remainder of this book looks at how these routines are used within an application.

Program Resources

Files with a file type of ".PRG" are program files. This is how GEM knows that this file can be executed. You may have noticed files with a file type of ".RSC" that have the same file name as a program file. This companion file is called a *resource file*. A resource file contains such items as the program's menu, dialog and alert boxes, icons, and any other graphic images that the program uses. Each item in a resource file is called an *object*. The term object is used quite loosely in the GEM documentation. The word object is a vague reference to anything from a simple string of text to an inventory entry form. For example, the menu title Desk, the menu title File, the Quit option under the File menu, and the icons on the desktop are objects and can be held in the resource file. The resource file holds these objects in an organized fashion to make them accessible to your application program.

Object Trees

The objects are organized in what is called a *tree* data structure. A tree consists of one object linked to several other objects, called the *children*. Any child may have children of its own, and the children's children can have children. The basic premise of the tree is that any item in the tree can have 0 or more children and 0 or 1 *parent*. This is the structure of the tree used in the resource file. Other tree structures exist but do not concern us here.

Figure 9-1 shows a tree for the basic menu used in a program. Notice that *only one* object has no parent. This object "Main Menu," is called the *root*. The root is used as the starting point for the entire tree. In the figure, the root has two children, "Desk" and "File," which are titles in the menu bar. Object File has one child, the option "Quit." Object Desk has eight children: six desk accessories, a place for your information message, and a disable dashed line. All boxes shown in the figure are considered to be individual objects. In this tree, the objects are just text. In general, any mix of object types may be included in a single tree.

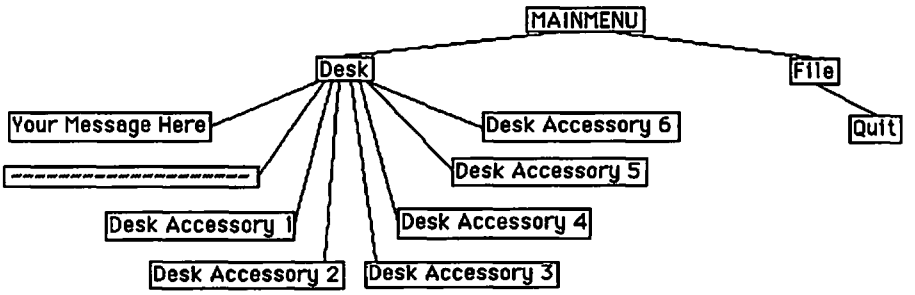


Figure 9-1 Sample Menu Object Tree

Figure 9-2 shows an example of a more complex resource tree. At the desktop, the Options menu has a Set Preferences item. When this item is selected, a dialog box appears. The resource tree of Figure 9-2 shows the tree associated with this dialog box. The root of the tree is the Set Preferences dialog box itself. In other words, the root object contains information that informs the AES that the tree is to draw a dialog box. Each of the text lines, "SET PREFERENCES," "Confirm Deletes:," "Confirm Copies:," and "Set Screen Resolution:" are text objects. The "OK" and "Cancel" objects are *exit buttons*. You can press these buttons by placing the mouse over either one and pressing the mouse button. They are called exit buttons because when you press either one, you signal that you want to exit the dialog box and continue with the program. The OK and Cancel buttons are children of the dialog box.

Three invisible boxes are children of the dialog box. The resource tree can hold many different types of objects. Invisible box 1 is the parent to the top yes/no button combination (see the dialog box on your desktop), invisible box 2 is the parent to the bottom yes/no button combination, and the third invisible box surrounds the

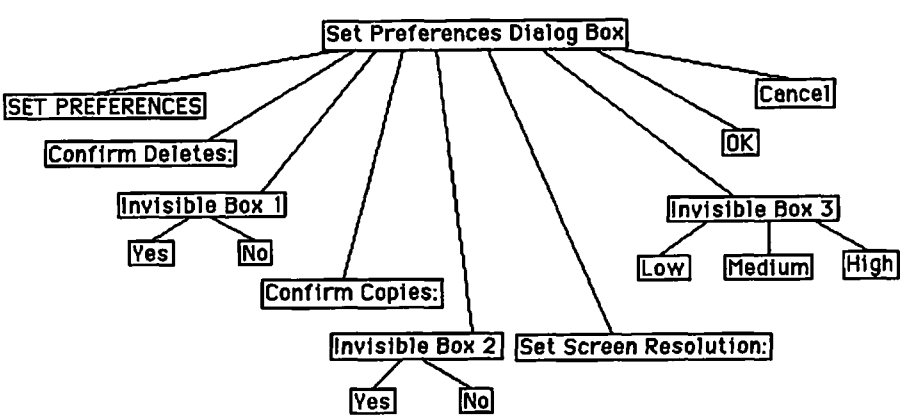


Figure 9-2 Set Preferences Dialog Box Object Tree

low/medium/high button selections. Each button in these sets of buttons is called a *radio button*. Much like the mechanical buttons on a car radio, only one button from each set may be selected at a time. For example, the screen resolution may be set only to low, medium, or high. You could not have a low-medium resolution set, just as you could not have yes and no confirmation of file deletions. When you press one button, the AES automatically deselects any other button. The AES knows which buttons belong to the same set because radio buttons of the same set have the same parent. Hence the need for the invisible boxes; each set of radio buttons has its own parent.

Object Structures

The OBJECT Structure

Figures 9-1 and 9-2 show the logical arrangement of a tree. This is how a human would visualize the tree. In a program, the tree's layout and information about each object must be represented in memory. In GEM this is done through the use of several C structures. Each entry in the tree is called an object and is represented by a structure called OBJECT (see Figure 9-3).

```
typedef struct object {
    WORD        ob_next;
    WORD        ob_head;
    WORD        ob_tail;
    WORD        ob_type;
    WORD        ob_flags;
    WORD        ob_state;
    CHAR        *ob_spec;
    WORD        ob_x;
    WORD        ob_y;
    WORD        ob_width;
    WORD        ob_height;
} OBJECT;
```

Figure 9-3 The OBJECT Structure

To represent the tree in memory, all the objects are placed in an array. The OBJECT structure field **ob_next** holds the index of the object's next sibling. Therefore, the **ob_next** field of the SET PREFERENCES object in Figure 9-2 has the array index for the Confirm Deletes: object. The field **ob_head** has the index to the object's first child. Thus, the **ob_head** field of the Set Screen Resolution: object has the index for the Low radio button object. The **ob_tail** field for the Set

Screen Resolution: object has the array index for the High radio button object, which is the last child of the parent. Therefore, one array is used to hold all the objects for a particular tree. By retrieving the array index for the children or siblings, your program can traverse through all objects in the tree.

The field **ob_x, ob_y, ob_width,** and **ob_height** determine the placement and size of the object on the screen. All values are measured in pixels. The x and y coordinates are *relative* to the object's parents. If the object is the root, the coordinates are screen coordinates. For example, if the root object of the dialog box has x and y coordinates of (200, 100), the upper left corner of the dialog box is placed at screen coordinates of (200, 100). Now assume that the SET PREFERENCES object has x and y coordinates of (20, 20). The string "SET PREFERENCES" is placed 20 pixels to the right and 20 pixels down from the upper left corner of the dialog box. This corresponds to screen coordinates of (220,120). This allows you to move the root object to any point on the screen and have all its contents move accordingly.

The **ob_state** field contains the current state of the object. The state of the object determines how the object is drawn and what functions it can perform depending upon what the object represents. The states are shown in Table 9-1 as they are defined in header file **obdefs.h.**

Table 9-1: Object State Definitions

Constant Name	*Value*
NORMAL	0X00
SELECTED	0X01
CROSSED	0X02
CHECKED	0X04
DISABLED	0X08
OUTLINED	0X10
SHADOWED	0X20

The NORMAL state indicates that the object is drawn using the normal foreground and background colors. The SELECTED state indicates that the object is drawn with the foreground and background colors reversed. CROSSED tells the AES that an "X" is to be drawn over the entire object as if it were crossed out. CHECKED says that a check mark is drawn at the leftmost edge (usually used for text-based objects). An object is DISABLED means that the object is drawn in a half-intensity mode. OUTLINED specified that an outline is to appear around a boxed object (only for objects in boxes). SHADOWED indi-

cates that the object (again usually a box) is drawn with a dropped shadow, meaning that the right and bottom edges are slightly thicker.

As you can see in Table 9-1, each object state indicates a particular bit setting. Therefore, various states can be combined such as CHECKED and SELECTED. NORMAL is when all bits are 0 or no states are active. Of course, some combinations such as SELECTED and DISABLED do not make sense and should not be set together. A bit is considered set when that particular bit has a value of 1.

The **ob_flag** field of the OBJECT structure is similar to the **ob_state** field. The various flags indicate the particular attribute(s) set for the object. Table 9-2 lists the object flag definitions.

Table 9-2: Object Flags

Constant Name	*Value*
NONE	0X000
SELECTABLE	0X001
DEFAULT	0X002
EXIT	0X004
EDITABLE	0X008
RBUTTON	0X010
LASTOB	0X020
TOUCHEXIT	0X040
HIDETREE	0X080
INDIRECT	0X100

The NONE flag indicates that there are special attributes for this object. SELECTABLE says that the object may be selected, usually indicating some kind of button or a box. DEFAULT is a flag for the Form library. Usually in the case of a dialog box, the DEFAULT flag is attached to one of the exit buttons. When the user enters a carriage return, this object is automatically selected as the exit object. Only one DEFAULT object is available in any one dialog box. The EXIT flag indicates to the Form library that when the user clicks the mouse on this object, the exit condition has been achieved and the Form library exits the dialog box. EDITABLE means that the user can edit the object in some way as specified within the application. RBUTTON specifies that the object is a radio button. LASTOB is a flag that indicates the last object is the object tree. If you build your own trees, you should use this flag. If you use another program to build your resource files (as is done for the programs in this book), you don't need to worry about setting this flag. TOUCHEXIT tells the Form library that when the user clicks this object, the form is completed and can be exited. HIDETREE makes this object and all its children

"invisible." There are a number of AES routines that locate or draw objects. If the HIDETREE flag is set, these functions are not able to draw or locate the object *or* any of its children. INDIRECT indicates that the value of the **ob_spec** field is a pointer to the actual **ob_spec**. This lets you change the **ob_spec** field by changing the pointer instead of constructing a new **ob_spec** field.

The **ob_type** field of the OBJECT structure provides the type of the object. The AES routines use this type information when processing the object. For example, if the type indicates a box, the AES draws a rectangle. If the type is a boxed string, the AES draws the text and encloses it in a box. Table 9-3 lists the defined object types and their values.

Table 9-3: Object Types

Constant Name	*Value*
G_BOX	20
G_TEXT	21
G_BOXTEXT	22
G_IMAGE	23
G_PROGDEF	24
G_IBOX	25
G_BUTTON	26
G_BOXCHAR	27
G_STRING	28
G_FTEXT	29
G_FBOXTEXT	30
G_ICON	31
G_TITLE	32

The type of the object also determines the contents of the **ob_spec** field. The **ob_spec** field may contain either the data layout shown in Figure 9-4 or a pointer to an additional information structure. Object types G_BOX, G_IBOX, and G_BOXCHAR all use the **ob_spec** layout shown in Figure 9-4. The high byte of the high word contains a character. The low byte of the high word contains a border thickness. The low word contains the color, writing mode, and fill pattern for the box.

The character portion of the **ob_spec** field is used only by type **G_BOXCHAR**, which is a single character enclosed in a box. The character portion contains the ASCII value (see Appendix C) of the character to be displayed. The border thickness applies to all boxes. If the border thickness is 0, the thickness is 0 and the box is invisible. If the border thickness is a positive value from 1 through 127, the thickness is measured from the edge of the box inward. Thus the

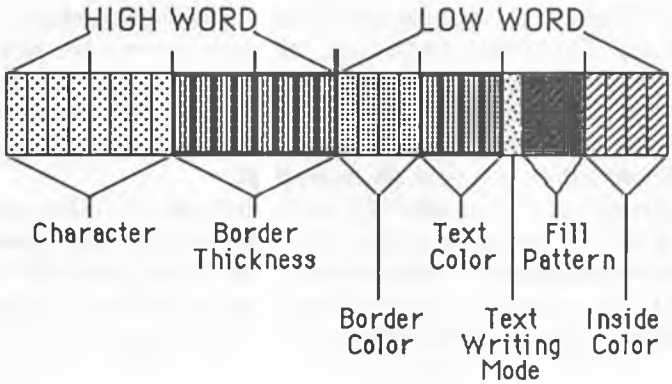


Figure 9-4 The ob_spec Field Data Layout

outside dimensions of the box remain the same and the border grows toward the center of the box. If the border thickness is a negative value from −1 through −127, the thickness is measured outward from the object's edge. Although it is not mentioned in the documentation, it is assumed that these values are measured in pixel units. The colors for a particular box use four bits.

The writing mode of the **ob_spec** field is a single bit. If it is set to 0, text is written in transparent mode. If the bit is set to 1, text is written in replace mode. The last section of the **ob_spec** field is the fill pattern. The fill pattern is a 3-bit value. If the three bits are all 0, the fill is hollow. If three bits are all 1 (evaluating to 7), the fill pattern is a solid fill. Values 1 through 6 indicate patterns of increasing darkness. The actual patterns used depend on the output device but the increase in density is guaranteed.

Object type G_BOX simply indicates a graphic box that may have the attributes indicated by the fields in the object structure. Object G_BOXCHAR is a box that contains one character of text. Object type G_IBOX is considered an invisible box. The fill pattern and internal color are ignored. The color and border thickness may be used. A border thickness of 0 makes the box invisible. One purpose of an invisible box is to provide a parent for a set of radio buttons.

Object type G_BUTTON is a graphic text object centered in a box. Type G_STRING (no comments from the back row, please) is simply a string of graphic text. Type G_TITLE is a text string used in menu titles. Each of these three types uses the **ob_spec** field as a pointer to a text string.

The remaining objects require more information than can be stored in the OBJECT structure. In these cases, the **ob_spec** field is used as a pointer to a secondary structure. The layout of the secondary

structure depends upon the type of object described. These additional structures are called TEDINFO, ICONBLK, BITBLK, APPLBLK, and PARMBLK.

The TEDINFO Structure

Types G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOXTEXT all use the **ob_spec** field as a pointer to a TEDINFO structure (see Figure 9-5). The TEDINFO structure is used with objects that have editable text. This is text that can be altered by the user during the course of the program. The AES routines use the TEDINFO structure to relieve the user of many of the tasks associated with data entry. The TEDINFO structure used in conjunction with the AES routines can provide automatic data entry, data validation, default values, and format templates.

```
typedef struct text_edinfo {
    char        *te_ptext;
    char        *te_ptmplt;
    char        *te_pvalid;
    WORD        te_font;
    WORD        te_junk1;
    WORD        te_just;
    WORD        te_color;
    WORD        te_junk2;
    WORD        te_thickness;
    WORD        te_txtlen;
    WORD        te_templen;
} TEDINFO;
```

Figure 9-5 The TEDINFO Structure

The field **te_ptext** is a pointer to the storage location for the entered text. If this storage location already has a string in it, this string is displayed as the default value. The GEM documentation states that if the first character in the string is the "@" symbol, the field is considered to be blank. Any remaining characters in this string are merely used as placeholders. For example, the string "@xyzpdq" would show seven blank spaces. However, this feature did not work properly on the author's sytem.

The **te_ptmplt** field is used for those object types that allow the user to enter data. These objects are indicated by the EDITABLE object flag setting. This TEDINFO field is used as a pointer to a text string template. The template contains the format of text. Any underscores

in the template are displayed as blanks and represent the editable characters of the field. For example, consider this template:

Date:__/__/__

This template indicates that there are three areas for data to be entered for a total of six characters. When this template is drawn on the screen, the user is only able to change the characters where the underscores are located. All other characters are shown as indicated.

The field **te_pvalid** is also a pointer to a text string. This string is used to validate any entered text. The characters contained in the validation string evaluate as follows:

the digit "9" allows only the digits 0 through 9 to occur at this location.
an "A" allows only spaces and upper-case letters to be entered.
an "a" allows upper- and lower-case letters and the space.
an "N" allows 0–9, A–Z, and the space.
an "n" allows 0–9, A–Z, a–z, and the space.
an "F" allows all valid filename characters plus "?", "*", and ":". This type of validation is used when a filename with a drive name and wildcard characters should be entered.
a "P" allows any pathname characters plus "\", ":", "?", and "*". This is used to locate files in other directories.
a "p" allows all valid pathname characters, plus "\" and ":". This allows the user to enter a specific file name but not the wildcard characters.
and an "X" allows any character to be entered.

By using the **te_pvalid** field, the AES routines can automatically perform some preliminary data verification. For example, the validation string for the date entry from above is "999999". This allows any six-digit entry to be placed in the **te_ptext** string. From this string, you can verify that a valid date has been entered. The advantage of using the validation string is that you do not have to worry about improper characters being entered. Note that the nonunderscore characters of the template field are not included in the validation string.

The field **te_font** indicates the font to be used for drawing the text. The value 3 means the system font is to be used (the font used for menus and dialog boxes). The value 5 indicates that the smaller font is to be used (as in icons). Field **te_junk1** (may also be called **te_resvd1**) is a reserved space. The **te_just** field is a word that indicates the type of text justification to be used when displaying the string. This justification occurs within the object as specified by the **ob_width** field in the OBJECT structure. A value of 0 is left-justified

text, 1 is right-justified, and 2 is centered. These values have defined constants labelled TE_LEFT, TE_RIGHT, and TE_CNTR, respectively.

The field **te_color** indicates the color and the pattern of objects in boxes, specifically G_BOXTEXT and G_FBOXTEXT. The value is divided into the various subfields as shown in the low word of Figure 9-4. The **te_junk2** (or **te_resvd2**) field is reserved. The field **te_thickness** indicates the thickness in pixels of the border of a box. The border thickness is computed in the same way as in the high word of Figure 9-4. Field **te_txtlen** is the length of the string pointed to by **te_ptext**, and field **te_tmplen** is the length of the string pointed to by **te_ptmplt**.

The AES uses the TEDINFO structure with the object types G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOXTEXT. Object type G_TEXT is simply graphic text, which is a string that can have the various attributes supplied in the TEDINFO structure. For a G_TEXT object, the **ob_spec** field of the OBJECT structure points to a TEDINFO structure. The field **te_ptext** of the TEDINFO structure points to the text string to be displayed. The G_BOXTEXT object is a rectangle containing graphic text. Again, the **ob_spec** field points to a TEDINFO structure that contains a pointer in the **te_ptext** field to the text string. The TEDINFO structure also contains the attributes for the box. The G_FTEXT object is formatted graphic text. Its **ob_spec** field points to a TEDINFO structure. The TEDINFO structure has a pointer to a text string in the **te_ptext** field and uses the template pointed to by **te_ptmplt** to format the text. The G_FBOXTEXT object is similar to the G_FTEXT object with the addition of a rectangle surrounding the text.

The ICONBLK Structure

Object type G_ICON indicates that the object describes an *icon*. An icon is a graphics figure used to represent some item in the computer system such as the disk drive for trash can on the desktop. The **ob_spec** field for a G_ICON object contains a pointer to the ICONBLK structure (see Figure 9-6. The ICONBLK structure holds data that defines an icon. The **ib_pmask** field is a pointer to an array of words representing the *mask bit image* of the icon. This is similar to the masking concept used in program BOUNCE. The **Ib_pdata** field is a pointer to an array of words representing the *data bit image* of the icon itself (like the ball array in BOUNCE). Field **ib_ptext** is a pointer to the icon's text, such as the word "TRASH" under the trash can. The **ib_char** field is a word containing the character to be drawn on the icon, like the drive letter on the floppy disk. The text and character fields may be empty and are not required. The field **ib_xchar** is the x

```
typedef struct icon_block {
    WORD        *ib_pmask;
    WORD        *ib_pdata;
    char        *ib_ptext;
    WORD        ib_char;
    WORD        ib_xchar;
    WORD        ib_ychar;
    WORD        ib_xicon;
    WORD        ib_yicon;
    WORD        ib_wicon;
    WORD        ib_hicon;
    WORD        ib_xtext;
    WORD        ib_ytext;
    WORD        ib_wtext;
    WORD        ib_htext;
} ICONBLK;
```

Figure 9-6 The ICONBLK Structure

coordinate in pixels of the character, and field **ib_ychar** is the y coordinate of the character. Again, the coordinates are relative to the upper left corner of the icon. The **ib_xicon** and **ib_yicon** fields are the x and y source coordinates of the icon, used as in the raster copy functions of the VDI. Field **ib_wicon** contains the width of the icon. As with the rasters, this width value must be divisible by 16 (an integral word width). The **ib_hicon** field contains the height of the icon. The **ib_xtext** and **ib_ytext** fields are the x and y coordinates of the icons's text. The **ib_wtext** and **ib_htext** are the width and height of a rectangle for the icon text. The text is centered within this rectangle. All size and position values are measured in pixels.

The BITBLK Structure

Object type G_IMAGE is a bit image object just like a raster. The **ob_spec** field for a G_IMAGE object points to a BITBLK structure as shown in Figure 9-7. The **bi_pdata** field of this structure points to an array of words containing the bit image (that is, raster). Field **bi_wb** is the width of the **bi_pdata** array in *bytes*. Because the **bi_pdata** array is made of words, the width must be an even-numbered value because there are two bytes per word. The **bi_hl** field is a word containing the height of the bit block in pixels. The **bi_x** is the source x coordinate relative to the **bi_pdata** array, and **bi_y** is the y source coordinate relative to that array. These values have the same use as the source coordinates used in the raster copy functions of the VDI. The field **bi_color** is a word containing the color that the AES uses to display the image. The color index values are shown in Table 9-4.

```
typedef struct bit_block {
    WORD        *bi_pdata;
    WORD        bi_wb;
    WORD        bi_hl;
    WORD        bi_x;
    WORD        bi_y;
    WORD        bi_color;
} BITBLK;
```

Figure 9-7 The BITBLK Structure

Table 9-4: Object Color Index Values

Constant Name	*Value*
WHITE	0
BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5
YELLOW	6
MAGENTA	7
LWHITE	8
LBLACK	9
LRED	10
LGREEN	11
LBLUE	12
LCYAN	13
LYELLOW	14
LMAGENTA	15

The APPLBLK and PARMBLK Structures

Object type G_PROGDEF is a programmer-defined object. When a program tells the AES to draw an object or object tree, the AES traverses through the tree and draws each object and its children (if they are not flagged as being hidden). For each object type, the AES has a routine to draw that particular object on the screen. For example, object type G_BOXTEXT has a routine that draws a box and then draws text within the box. Object type G_PROGDEF tells the AES that this object is not one of the standard object types and requires a special handling routine. The location of this custom routine is provided in the **ab_code** field of the APPLBLK structure (see Figure 9-8). This field contains the pointer to the custom routine. The other field in the APPLBLK structure, called **ab_parm**, is a value of

```
typedef struct appl_blk {
    int         (*ab_code) ( );
    long        ab_parm;
} APPLBLK;
```

Figure 9-8 The APPLBLK Structure

type *long*. This is used as a four-byte parameter to the custom routine. A program can divide these four bytes in any combination: four characters (one byte each), two integers (two bytes each), one integer and two characters, a long integer (four bytes), or a pointer (four bytes).

When the AES draws or changes an object, it needs much more information than can be supplied in just four bytes. This information includes the location of the object tree, the index of the object being drawn or changed, the location of the object, and so on. When the AES encounters a custom object, it calls the routine specified by the **ab_code** field and provides this routine (as a parameter) with the additional information in a PARMBLK structure (see Figure 9-9). In this structure, the **pb_tree** field is a pointer to the object tree that contains the object to be operated upon. The **pb_obj** field contains the index of that object within the tree. Since object trees are stored as arrays, the index is the element number of the object. The **pb_prevstate** field is the previous state of the object to be changed, and the **pb_currstate** field is the new state of the object. When **pb_prevstate** and **pb_currstate** are the same, the application draws the object and does not change it. Fields **pb_x** and **pb_y** are the x and y coordinates of the upper left corner of a rectangle defining the location of the object. Fields **pb_w** and **pb_h** are the width and height of the rectangle. The next four fields, **pb_xc**, **pb_yc**, **pb_wc**, and **pb_hc**, determine the location and size of the clipping rectangle. The x and y coordinates are the upper left corner of the rectangle and are given by **pb_xc** and **pb_yc**. The width and height are given by **pb_wc** and

```
typedef struct parm_blk {
    OBJECT      *pb_tree;
    WORD        pb_obj;
    WORD        pb_prevstate;
    WORD        pb_currstate;
    WORD        pb_x, pb_y, pb_w, pb_h;
    WORD        pb_xc, pb_yc, pb_wc, pb_hc;
    long        pb_parm;
} PARMBLK;
```

Figure 9-9 The PARMBLK Structure

pb_hc. The clipping rectangle is used here just as in the VDI. Anything outside of the clipping rectangle is not drawn. The **pb_parm** field is set to the same value as the **ab_parm** field in the APPLBLK structure. All coordinate values for this structure refer to the physical screen. In other words, the x and y coordinates for the object and the clipping rectangle are the x and y coordinates on the screen. They are not relative to the parent object.

The information about objects and their related structures is a very powerful tool in the program; this is done by defining an object tree for a dialog box, menu, icon, or other object you want displayed on the screen. Pass this tree to the appropriate AES routine, and it appears on the screen. Don't be overwhelmed by the complexity of creating an image on the screen, even an image as simple as a dialog box. The AES contains many routines that assist you in handling object tree management like adding, deleting, reordering, and changing objects. Other AES routines perform the drawing of objects, data entry, data verification, as well as the user interaction involved with the selection of boxes and icons.

The Resource Construction Program

The use of objects in a GEM application is a very basic requirement. However, even with assistance from the AES routines, the creation of object trees is a quite cumbersome task. To relieve your program of the tedium of creating the data and making sure that is correct, there is a program provided that creates the resource file to be used by the program. The Megamax compiler comes with a program called MMRCP, which allows you to create a resource file. The Atari developer's kit also comes with a program called RCS that performs the same task. The two programs produce a resource file. Since the objects have a predefined format, you may use either program to create the same resource file.

At the time of this writing, the version of the Megamax MMRCP program had a problem when writing out the resource file. The discussion of the resource program therefore concerns the Atari/Digital Research RCS program. The concepts discussed here are the same for both programs. However, the operation of each program is slightly different.

To do the remaining programs in this book, you need a resource editor program or some method of creating the resource files to be used by the programs. Therefore, at this point, a brief tutorial for the RCS resource construction program is provided.

First, read through the manual for the resource editing program

you are using. Familiarize yourself with its features and general operation. Then load this program on your Atari system, and execute it. For the RCS program, you see a menu bar, a window called "RESOURCE PARTBOX," a window called "RESOURCE CONSTRUCTION SET," a clipboard, and a trash can. The RESOURCE PARTBOX window contains icons for the various types of objects trees that you can create. These are a menu, a dialog box, an alert box, a free tree, and a tree of unknown type. A free tree is simply a tree that contains a set of objects. A tree of this type is usually just to format a display screen. The unknown tree type is used when the resource editor program is changing an existing resource file and does not know what type of tree to use. For example, if you read in the resource file for the resource editor (file RCS.RSC for the RCS program), you see a bunch of trees of unknown type. If you are not using the RCS program, your screen should have a menu bar, some portion of the screen containing the tree type icons, and a portion of the screen that is used for editing the resource file.

You are now going to create a dialog box. To begin, use the File menu to open a new file. For the RCS program, selecting the New option causes the workspace to be cleared. Any program may contain any number of object trees (memory space permitting). For example, the RCS program has a tree for the menu bar and many dialog boxes such as a dialog box for the Program Information option in the Desk menu, and a dialog box to name an object. Each of these trees is given a root. To "plant" (create) a tree, drag the appropriate tree icon from the part box to the work area. For the dialog box example used here, drag a dialog tree to the work area. At this point, the RCS program displays its own dialog box requesting the name of the tree with a default name of TREE1. Change this name to ENTRYBOX. If your resource editor did not request a name, select this tree by clicking on it once. Then use the Name option in the menu (or something equivalent) to set the name of this dialog box object tree.

Now that you have set the root of the tree, you need to define what the contents of the dialog box will look like. This is done by placing objects into the dialog box. Double-click the ENTRYBOX icon. This opens a window for the dialog box and sets the parts box to contain the objects that you may want to use. The contents of the parts box vary from program to program. In general, it contains such objects as a button, a string, formatted text, editable fields, and boxed text. Just as in planting a tree, you place objects in the dialog box by dragging them from the parts box to the work area. The dialog box you are now creating is a sample survey form that demonstrates how to use editable fields, templates, data validation, buttons, radio buttons, and default exit conditions (see Figure 9-10).

PERSONAL DATA SHEET

Last Name:_____ Date of Birth:__/__/__
First Name:_____ # of Years at
Current Address:__

Yearly Income (Check One)
- Under $14,999 [ ]
- $15,000 - $24,999 [ ]
- $25,000 - $49,999 [ ]
- $50,000 and over [ ]

Hobbies (Check All That Apply)
- Skiing [ ]
- Tennis [ ]
- Music [ ]
- Swimming [ ]

[OK] [Quit] [Clear Form]

Figure 9-10 ENTRYBOX Dialog Box

The title of the dialog box, "PERSONAL DATA SHEET," is just a string object. To create the title, take a string object from the dialog part box and place it at the location of the dialog box title. You now have a text string that says "STRING" at this position. Double-click this object, and the resource editor program opens up another of its own dialog boxes. This dialog box allows you to set the various object flags and states for this object. At the bottom of the box is a line that says:

TEXT: STRING_____

Whatever you enter in this field becomes the text that is displayed for this object. Press the Escape key to clear the text field and enter the words "PERSONAL DATA SHEET." Click the OK button. The dialog box disappears and the new text of the title is shown on the box you are creating.

To create the editable text fields for the first and last name data, use the unboxed edit object. This is the object that says "EDIT:_____." Drag this object onto your dialog box, and place it at the location of the last name entry field (refer to Figure 9-10). Double-click this new object, and another dialog box appears. This box has object flag and state selections, background color and intensity, border color and size, character color, font, justification, writing mode (called Rule in the RCS), and text fields. Because this particular object is not a boxed object, the background and border selections have no effect on how this object is displayed. These options are included here because this dialog box is also used for the boxed EDIT:_____ object.

Note that the object flag button labeled "EDITABLE" is selected (shown in reverse colors). This is the default flag setting for the

EDIT:_____ object and indicates that this object is editable. If this flag is not set when your program activates this dialog box, the AES does not let the user edit this field. For the dialog box you are creating, change the PTMPLT line at the bottom to show:

Last Name:~~~~~~~~~~~~~~~~~~~~

Use the up, down, left, and right arrows on the keyboard to move the text cursor. When used in the resource editor program, the tilde character (~) indicates an underscore. This is done because the underscore is already used to show what portions of the field may be edited. If you enter the underscore, there is no way of knowing what portion of the field is your entry and what portion is displayed by the program.

The PVALID line is the validation text. This line must match the format of PTMPLT line in that you must place the validation characters at the exact position of the entry. In other words, wherever you do not want data to be entered, place a tilde. Wherever you want data to be entered, place the appropriate data validation character (see the TEDINFO structure above). For this dialog box, place tildes under the "Last Name:" portion of the PTMPLT line. Then put a capital X under each tilde in the PTMPLT line. The format of these fields may vary slightly among resource editor programs; however, the general layout remains the same.

The PTEXT field is the default text to be displayed when the program is run. When the AES displays this dialog box, the string shown on this line appears as the current value of the entered date. The PTEXT field also determines the maximum number of characters the user may enter. This field is formatted with tildes used for places where no text is entered and underscores (_) where text will be entered. When the AES draws this dialog box, it draws the PTMPLT field replacing the tildes with the text from the PTEXT field. In this example dialog box, you need 11 tildes followed by 15 underscores to match the format of the PTMPLT field. Enter these values for the PTMPLT, PVALID, and PTEXT fields.

PTMPLT>Last Name:~~~~~~~~~~~~~~~~~~~~
PVALID>~~~~~~~~~~XXXXXXXXXXXXXXXXX
PTEXT>~~~~~~~~~~_____

Click OK and when the object's dialog box disappears, you see:

Last Name:_____

which is what is shown when the AES draws this object.

Logically, you might think that the PTMPLT field should control the format of the object, the PVALID field should have only the validation portion of the string, and the PTEXT field should have only

the default text. In the RCS program, this is not the case. The **PVALID** and **PTEXT** fields must match the format of the **PTMPLT** field. Other resource editor programs may have different requirements.

Take another edit field and create a first name entry field in the same manner you created the last name entry field. The first name entry field has the following values:

```
PTMPLT>First Name: ~~~~~~~~~~~~~~~~~~~~
 PVALID>~~~~~~~~~~~XXXXXXXXXXXXXXXXXX
  PTEXT>~~~~~~~~~~~____________________
```

For the date of birth entry, the values are these:

```
PTMPLT>Date of Birth:~~ / ~~ / ~~
 PVALID>~~~~~~~~~~~~~99~99~99
  PTEXT>~~~~~~~~~~~~__~__~__
```

Notice that tildes are placeholders for noneditable characters and that the validation string uses the character "9" to allow only numeric values.

The text "# of Years at" is a STRING object like the box title. Create this string in the same manner as before, then make the "Current Address" entry field with these values:

```
PTMPLT>Current Address:~~
 PVALID>~~~~~~~~~~~~~~~~~99
  PTEXT>~~~~~~~~~~~~~~~~~__
```

This field allows a two-digit value to be entered.

The remaining objects in this dialog box are strings and buttons. The yearly income items are all STRING objects as are the hobby items. Create the STRING objects for the yearly income selections and the hobby selections including the headings "Yearly Income" and "Hobbies."

The small objects that say "Check One" and "Check All That Apply" are TEXT, not STRING, objects. A STRING object allows you only to enter a string and set some of the object flags and states. A TEXT object, in addition to setting the text, flags, and states, also allows you to set the font size, text justification, colors, and writing mode. Because small lettering is used in the ENTRYBOX dialog box, the TEXT object is required. Three TEXT objects are needed: one for "Check One," one for "Check All," and one for "That Apply." Because "Check All That Apply" is split across two lines, two TEXT objects are required. Drag three TEXT objects from the parts box to the work area and place them at the appropriate locations. Double-click the TEXT object for the "Check One" object and a dialog box is displayed. This is the same dialog box used for the editable text fields. With a TEXT object, however, the PTMPLT and PVALID fields have no effect. Only

the PTEXT field is used, and it should be set to read "Check One." Next select the small font and click OK. Repeat this procedure for the other two TEXT objects, "Check All" and "That Apply."

By now you may have noticed that whenever you place an object in your dialog box, it appears to line up on an invisible grid. When you are editing a dialog box, all objects contained in it are character-aligned. This means that the dialog box is divided into character cells. The start of an object must lie in one of these cells. You cannot place an object half-way between cells, and the length of an object must be a whole number of cells. This relieves you of having to align the text by hand. If you want more control over where you can place an object, use the free type tree.

At this point, you now have a form that has all the text in it. You now need to create the four boxes for yearly income, the four boxes for hobbies, and the three buttons at the bottom of the form. The yearly income selection uses radio buttons because only one income level at a time is allowed. As mentioned earlier, radio buttons require a common parent. In other words, all radio buttons in a set must be children of the same parent. To create a common parent, drag the single outlined box from the parts box to the area under the "Check One" object and next to the "Under $14,999" object. You now have a horizontal box where you want a vertical box. Therefore, you need to resize this object. To do this, select the box object by clicking on it once. This causes the object to be shown in reverse colors. Now place the tip of the mouse arrow cursor at the bottom right corner. When you press the mouse button, the arrow cursor changes to the finger cursor. This may take a few attempts to hit the exact spot on the object for sizing. With the finger cursor, you can resize the object just as you would resize a window. Releasing the mouse button sets the new object size. You can move an object by placing the arrow cursor at the center of the object and pressing the mouse button. The arrow cursor changes to the open hand cursor, and you can drag the object to its new location. Resize the common parent box so that it is about three characters wide and four lines tall under the "Check One" object (see the shaded area Figure 9-10). Move the box if required.

You now have the box to be used as the common parent. You now need to add the four children. An object is considered the child of another object if the parent object completely encloses the child. For example, all the objects you have placed in the dialog box so far are children of the dialog box because the dialog box contains these objects. To create a small box, take a single outlined box from the parts box and place it in an empty area of the dialog box such as the upper left corner. Resize the new box so that it is one line high and one character wide. Drag this box so that it is under the "Check One" object on the same line as the "Under $14,999" object and fully

enclosed by the parent box you just created. If you are using the RCS program, you receive an alert box saying that this move rearranges the tree. What is happening is that you are taking a child of the dialog box (because it was contained only by the dialog box) and making it a child of the parent box. This changes how the tree looks. To verify that you want to complete this action, the RCS program warns you of the situation and allows you to cancel the move. In creating ENTRYBOX, you want to change the order of the tree. Create three more boxes for the remaining three income levels.

In the final form of the dialog box, you do not want the parent box to be visible. Therefore, double-click this box and change the border to be invisible (that is, no border). Click OK, and when ENTRYBOX is redisplayed, the parent box will be invisible.

You now have four small boxes surrounded by an invisible parent box. These four boxes must be turned into radio buttons. Double-click the top box. This causes a dialog box to be displayed. This dialog box allows you to set the flags, states, and text of a box. Click the SELECTABLE flag. This turns the box into a button the user can select with the mouse. Also click the RADIO BUTN flag to make the button into a radio button. The TEXT field remains blank because there is no text for this button. Click OK and that's all there is to making a radio button. Repeat this procedure for the other three boxes. The radio buttons are now ready.

The hobby selection also requires four selection or check boxes. These are not radio buttons because a person may have more than one hobby. Create four small boxes and place them in a column. For each box, double-click it and choose the SELECTABLE flag *only*. These boxes are now buttons. Alternatively, instead of using the single outlined boxes, you can use the BUTTON object that already has its SELECTABLE flag set.

The top portion of the form is now completely set. The next step is to set the three buttons at the bottom of the form. Take three BUTTON objects from the parts box, and put one in place for each button. At this point, you are probably running out of room on the screen. Click the full box (at the right end of the title bar) to change the size of the window. You can now resize the dialog box itself and move the buttons to their appropriate locations.

The OK button is an exit button. If the user selects this button, the AES stops processing the dialog box and returns control to the application program. Every dialog box must have some form of exit object; otherwise, the user can never get back to the program. The OK button is also the default exit selection so that if the user presses the RETURN key, the OK button is automatically selected. Double-click the button that becomes the OK button. Given the above parameters, the SELECTABLE, DEFAULT, and EXIT flags should be set in the

dialog box (SELECTABLE is already set for a BUTTON object). The TEXT field should be set to read "OK." Upon return to the ENTRYBOX dialog box that you are creating, you will notice that the OK button has a thicker outline than the other buttons on the form. The thick border indicates that the OK button is the default exit value.

Double-click the button that is used to quit. Change its text to read "QUIT" and select the EXIT flag (the SELECTABLE flag is already set). When ENTRYBOX is redisplayed, this button says QUIT and its border has a medium thickness. A button with medium thickness indicates an exit button that is not the default.

Finally, double-click the last button. Choose the EXIT flag, and change its text to read "Clear Form." This button is used if the person wishes to clear the form without further processing.

You have now created all the objects used by the ENTRYBOX dialog box. You could save this in a resource file and use the resource file in a program. However, there is one small problem. Many AES functions require that you know the index number of the objects you need to access. Depending upon the order in which you created the resource file, these index numbers vary. Therefore, if you edit the resource file and add or delete objects, the index numbers change. How do you know what index numbers to use?

Fortunately, the RCS program and the MMRCP program allow you to name the objects. For example, ENTRYBOX is the name of the tree used for the dialog box. The name of the object is then used as a defined constant within your program. When you save the resource file, the resource editor program writes the resource file and a C header file containing the names of the objects (objects not named are not included in the header file). If you save ENTRYBOX in its current state, you would get a resource file and a header file containing the statement:

```
#define    ENTRYBOX    0
```

The name ENTRYBOX is associated with the root of the dialog box (the box itself) and its index in the tree is element 0. By using the defined constant names and including the header file in your C program, you need not concern yourself with the actual index values when writing the program. Also, if you want to add more objects, simply do so and then recompile your program. The header file will contain the new index values for the constant names used by your program.

To name an object, click it once to select it. Then choose the Name selection from the Options menu (or whatever is appropriate for your resource editor program). For example, select the last name entry field and choose the Name selection. You get a dialog box that allows you to enter the name of the object and change the object type. Generally,

it is not a good idea to change the object type unless you really must. The Name field is blank. Enter LNAME. Object names should be in capital letters (which is a good idea since these names will be used for the defined constant names) and are limited to eight characters. The RCS program does not allow the underscore to be used in an object name; actually, entering an underscore caused the program to exit abnormally. This is probably a bug in the program. After entering the object name, click the OK button. That is all there is to naming an object. Now when the header file is written, there is also a statement defining the constant name LNAME. Repeat this naming procedure for the other objects in the dialog box as shown in Table 9-5. Note that not all the objects are named. Only those objects that are accessed by a program are named. The first name field should be called FNAME. Date of Birth should be called Birthday. Current address should be called HOMEAGE. Select the box associated with an income of less than $14,999. Name it INC1. Name the next, in order, INC2, INC3, and INC4. For the hobbies, name the box next to "skiing" as HOB1. The others are HOB2, HOB3, and HOB4. All the data items are now named. Finally, name the "OK" button OK. Name the Quit button QUIT and the Clear Form button CLEAR.

Table 9-5: Object Name for the ENTRYBOX Dialog Box

Object Field on Form	*Object Name*
Last Name	LNAME
First Name	FNAME
Date of Birth	BIRTHDAY
Current Address	HOMEAGE
Radio button for	
Under $14,999	INC1
$15,000–$24,999	INC2
$25,000–$49,999	INC3
$50,000 and over	INC4
Selection button for	
Skiing	HOB1
Tennis	HOB2
Music	HOB3
Swimming	HOB4
OK button	OK
QUIT button	QUIT
Clear Form button	CLEAR

After naming the objects, press the full box again to restore the window to its normal size. Then close the window for the ENTRYBOX dialog box. The program returns to the main window, which shows the dialog box icon for ENTRYBOX.

In the File menu, use the Save or Save As... option to save this resource file. Use the filename FORM.RSC. Exit from the resource editor program. When you return to the desktop, you see the file FORM.RSC in the directory, and two other new files: FORM.DEF and FORM.H. FORM.DEF is a definition file that is used by the resource editor program. It contains the names and basic layout of the resource file. If a .DEF file does not exist for a .RSC file, then when you try to edit the resource file, you get trees with the unknown tree type because the resource editor program does not know the layout of the file. File FORM.H is the header file to be included in your program. Double-click this file. GEM gives you a dialog box that says you can only print or display this document. Select the Show option to display the contents of the header file on the screen. You see a set of **#define** statements that list all the object names you entered. The number following the **#define** name is the index for that object in the tree.

The AES Review

With this introduction to the AES, you have learned all the basic concepts associated with writing programs for a GEM application. Because of the amount of information presented here, we are including a quick review of the more complex and necessary concepts.

You should intuitively understand how to use menus and windows. These are basic elements to using the Atari computer, so it is assumed that you have a working knowledge of these units. The first important concept to understand is the object. An object is simply a graphics unit that the AES can draw and possibly edit. Objects are logically grouped into units called trees. A tree describes a particular AES aspect such as a menu bar, a dialog box, or an alert box. A tree has a root object that generally describes the type of tree. Each object in a tree has one parent (except the root, which has no parents) and any number of children. The AES implements the tree data structure as an array. To access a particular object in an array, you must know the index number of the object in the tree.

Information about a particular object is held in an OBJECT structure. This structure contains data such as indices to the object's sibling, first child, last child, object type, state, flags, relative location on the screen, and a specification field. If more information is required by the particular object type, an additional structure is used. Of these additional structures, the one used most is the TEDINFO structure. This structure is used with objects that have editable text.

To relieve the program from the tedium of having to create object trees, a GEM development system usually comes with a resource file

editor program. This program allows the programmer to create and edit resource files which hold the object trees for a specific application. Most resource editor programs produce three output files. One file has a file type of .DEF and is used only by the resource editor program only to describe the layout of the resource file. The second file has the file type of .RSC and is the actual resource file. The last file is a C header file (file type .H), which contains defined constant names for the index of each named object.

At this point, you should experiment with the resource editor program. Try the various flag and state options for the different object types. Create a free tree, and place all of the objects in it. Then compare similar objects to see what makes them different. For example, the TEXT and STRING objects have different capabilities. Also play with the PTMPLT, PVALID, and PTEXT fields of an EDIT object. You will notice immediate changes with the PTMPLT and PVALID fields. The PTEXT field has no effect until you use the resource file in a program.

The set of sample programs in the next chapter should clarify how these objects are used and how the resource editor program interfaces with your application program. The first sample program, FORM, uses the dialog box just created. Then, we move on to the use of menus and interacting with the mouse to select objects in the menu. Finally, you create a program that allows you to display files on the screen.

CHAPTER TEN

Resourceful Programming

Previous chapters have covered what the AES does, its primary data structures (the object and object trees), and the resource editor program. How to create a dialog box, one of the several types of object trees that can be created with the resource editor program, has been demonstrated. This chapter shows how to use this information in an actual application.

Program FORM

Program FORM is a demonstration program that utilizes the dialog box created in the last chapter. With this program, you are able to experiment with some of the topics mentioned in Chapter 9. A few new overhead procedures are required by the AES. Take a look at function **main( )** in program FORM (see Listing 10-1). The initialize GEM Access section is the same as that of all other programs covered thus far. The application-specific routines start with a call to the function **rsrc_load( )**. This is an AES function that causes the AES to load a resource file into memory and keep track of the objects loaded. The basic process for an application using a resource file is to load the resource file into the AES memory, locate the tree (array) you want to use, and use the **rsrc_free( )** function to have the AES release the memory used for the resource file when the program ends. The **rsrc_load( )** function returns 1 if the load was successful or 0 if there was an error during loading. The system can crash if you try to access your resources without having the resource file loaded. In

Listing 10-1 Program FORM

```
/*********************************************************************
     FORM.C Dialog box and form demonstration program

     This program demonstrates the use of a dialog box to
     request information from the user.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max;                   /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

#include  "form.h"            /* resource header file */

char last_name[16],
     first_name[11],
     birth_date[7],
     home_age[3];

/*************************************************
     GEM-related Functions
*************************************************/
```

Listing 10-1 (continued)

```
WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}


set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}


/*************************************************
     Application Functions
*************************************************/


init_form()
/*************************************************
Function: Initialize text pointers for entry form.
Input:    None. Resource file must be loaded.
Output:   Sets form pointers.
*************************************************/
{
OBJECT    *box_addr;
```

Listing 10-1 (continued)

```
/* get address of dialog box */
     rsrc_gaddr(0, ENTRYBOX, &box_addr);

/* set addresses and lengths of text input */
     ((TEDINFO *)box_addr[LNAME].ob_spec)->te_ptext = last_name;
     ((TEDINFO *)box_addr[LNAME].ob_spec)->te_txtlen = 16;

     ((TEDINFO *)box_addr[FNAME].ob_spec)->te_ptext = first_name;
     ((TEDINFO *)box_addr[FNAME].ob_spec)->te_txtlen = 11;

     ((TEDINFO *)box_addr[BIRTHDAY].ob_spec)->te_ptext = birth_date;
     ((TEDINFO *)box_addr[BIRTHDAY].ob_spec)->te_txtlen = 7;

     ((TEDINFO *)box_addr[HOMEAGE].ob_spec)->te_ptext = home_age;
     ((TEDINFO *)box_addr[HOMEAGE].ob_spec)->te_txtlen = 3;

     return;
}

get_form()
/*************************************************
Function: Display dialog box to get data from user.
Input:    None. Resource file must be loaded.
Output:   Returns index of object used for exit.
          Dialog box objects selected and filled.
*************************************************/
{
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of dialog box */
     rsrc_gaddr(0, ENTRYBOX, &box_addr);

/* show initial text entry values */
     printf("\nLast Name  - Address: %8lx, Length: %2d\n",
          ((TEDINFO *)box_addr[LNAME].ob_spec)->te_ptext,
          ((TEDINFO *)box_addr[LNAME].ob_spec)->te_txtlen);
     printf("First Name - Address: %8lx, Length: %2d\n",
          ((TEDINFO *)box_addr[FNAME].ob_spec)->te_ptext,
          ((TEDINFO *)box_addr[FNAME].ob_spec)->te_txtlen);
     printf("Birthdate  - Address: %8lx, Length: %2d\n",
          ((TEDINFO *)box_addr[BIRTHDAY].ob_spec)->te_ptext,
          ((TEDINFO *)box_addr[BIRTHDAY].ob_spec)->te_txtlen);
     printf("Years      - Address: %8lx, Length: %2d\n",
          ((TEDINFO *)box_addr[HOMEAGE].ob_spec)->te_ptext,
          ((TEDINFO *)box_addr[HOMEAGE].ob_spec)->te_txtlen);
     Crawcin();
```

Listing 10-1 (continued)

```
/* clear entry screen */
     *(((TEDINFO *)box_addr[LNAME].ob_spec)->te_ptext) = 0;
     *(((TEDINFO *)box_addr[FNAME].ob_spec)->te_ptext) = 0;
     *(((TEDINFO *)box_addr[BIRTHDAY].ob_spec)->te_ptext) = 0;
     *(((TEDINFO *)box_addr[HOMEAGE].ob_spec)->te_ptext) = 0;
     box_addr[INC1].ob_state  = NORMAL;
     box_addr[INC2].ob_state  = NORMAL;
     box_addr[INC3].ob_state  = NORMAL;
     box_addr[INC4].ob_state  = NORMAL;
     box_addr[HOB1].ob_state  = NORMAL;
     box_addr[HOB2].ob_state  = NORMAL;
     box_addr[HOB3].ob_state  = NORMAL;
     box_addr[HOB4].ob_state  = NORMAL;

/* get size and location of a box centered on screen */
     form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
     smallx = xbox + (wbox / 2);
     smally = ybox + (hbox / 2);
     smallw = 0;
     smallh = 0;

/* reserve area on screen for box display */
     form_dial(FMD_START,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw an expanding box */
     form_dial(FMD_GROW,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw dialog box */
     objc_draw(box_addr, ENTRYBOX, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
     exit_object = form_do(box_addr, LNAME);

/* draw a shrinking box */
     form_dial(FMD_SHRINK,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
     form_dial(FMD_FINISH,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* unselect exit object for next time */
     box_addr[exit_object].ob_state = NORMAL;
```

Listing 10-1 (continued)

```
     return(exit_object);
}

show_info()
/*************************************************
Function: Show information entered into dialog box.
Input:    None. The dialog box objects must be selected.
Output:   None.
*************************************************/
{
OBJECT    *form_addr;

/* get base address of dialog box */
     rsrc_gaddr(0, ENTRYBOX, &form_addr);

/* display settings */
     v_clrwk(screen_vhandle);
     v_curhome(screen_vhandle);

     printf("Name                    : %s %s\n",
          ((TEDINFO *)form_addr[FNAME].ob_spec)->te_ptext,
          ((TEDINFO *)form_addr[LNAME].cb_spec)->te_ptext);
     printf("Date of Birth           : %s\n",
          ((TEDINFO *)form_addr[BIRTHDAY].ob_spec)->te_ptext);
     printf("Years at Current Address: %s\n",
          ((TEDINFO *)form_addr[HOMEAGE].ob_spec)->te_ptext);
     printf("Income Level            : ");
     if (form_addr[INC1].ob_state & SELECTED)
          printf("Under $15,000");
     else if (form_addr[INC2].ob_state & SELECTED)
          printf("$15,000 - $24,999");
     else if (form_addr[INC3].ob_state & SELECTED)
          printf("$25,000 - $49,999");
     else if (form_addr[INC4].ob_state & SELECTED)
          printf("$50,000 and over");

     printf("\nHobbies                 : ");
     if (form_addr[HOB1].ob_state & SELECTED)
          printf("Skiing   ");
     if (form_addr[HOB2].ob_state & SELECTED)
          printf("Tennis   ");
     if (form_addr[HOB3].ob_state & SELECTED)
          printf("Music   ");
     if (form_addr[HOB4].ob_state & SELECTED)
          printf("Swimming   ");

     printf("\n\nPress any key to continue\n");
     Craucin();
```

Listing 10-1 (continued)

```
    return;
}

/*************************************************
    Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

WORD ret_code;                      /* form exit code */

/*************************************************
    Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }

     screen_phandle =               /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Get screen attributes */

/*************************************************
    Application Specific Routines
*************************************************/

     if (!rsrc_load("FORM.RSC"))    /* trouble loading resource file */
     {
          form_alert(1, "[0][Could not load file FORM.RSC|Program terminating...][OK]");
          exit(1);
     }

/* allow user to skip form initialization */
     printf("\n\nPress 'I' to initialize form\n");
     if ( (Crawcin() & 0x7f) == 'I')
          init_form();
```

Listing 10-1 (continued)

```
    while ( (ret_code = get_form()) != QUIT)
    {
        if (ret_code != CLEAR)
            show_info();
    }

/**************************************************
    Program Clean-up and Exit
**************************************************/

    rsrc_free();                /* release memory for resources */
    v_clsvwk(screen_vhandle);   /* close workstation */
    appl_exit();                /* end program */
}
/**************************************************/
/**************/
```

FORM, the program uses the **rsrc_load( )** function to load the resource file. The result of this function is tested. If the resource file has not been loaded properly, the **form_alert( )** function is called to tell the user an error has occurred.

The **form_alert( )** function causes an alert box to be displayed. This function has two parameters. The first parameter indicates the default exit button for the form. A value of 0 means no default button. A value of 1, 2, or 3 indicates the first, second, or third button, respectively. The second parameter is a string defining the format of the alert box. This string is divided into three fields with each field surrounded by square brackets. The first field contains a digit for the icon type to use. Type 0 means no icon. Type 1 means the Note icon, which is an exclamation point. Type 2 is the Wait icon, which is a question mark. Type 3 is a stop sign called the Stop icon. The second field is the message text. The alert box can have up to five text lines, each with up to 40 characters. In this field, a vertical bar indicates the separation of one line from the next. In the program, the text "Could not load file FORM.RSC" forms the first line, and "Program terminating . . ." forms the second line. The third field of this second parameter indicates the text used in the exit buttons. The text for each button is separated by a vertical bar. The text for each button may contain no more than 20 characters. In FORM, there is only one exit button labeled "OK." The user has no choice but to end the program.

Once the resource file has been loaded, the program can begin its processing. The first process in FORM is to initialize the default values used in the dialog box. To assist you in experimenting with

the resource file and objects, FORM gives you the option of initializing the dialog box or skipping the initialization. Initialization can be used to set the storage size and location of the data to be entered. The TEDINFO structure has the **te_ptext** field, which points to the entered data and field **te_txtlen**, which contains the length of this data. In the resource editor program, the PTEXT field of an editable object was filled with underscores to set the length of the editable text (**te_txtlen**). In addition to setting the length, the underscores also allocate the memory space required to store the text string. The **te_ptext** field is only a pointer and does not know if that location is reserved for the entered string. If the space is not reserved, the AES starts to write over other data in memory. One method of avoiding this problem is to use the underscore in the resource editor program. The other method is to override the values created by the resource file by simply resetting the appropriate fields in the TEDINFO structure. For example, a program can set the **te_txtlen** field to whatever string length is required and can set the **te_ptext** field to point to a string or space allocated through **Malloc( )**. A word of warning: the **te_ptext** field must point to a memory location reserved for the entered string and the **te_txtlen** must have a value less than or equal to the amount of space reserved. If these two conditions are not met, you may alter portions of your program that should not be changed.

After the initialization is a loop that displays the dialog box and has the user enter the information. If the OK Exit button is used, the data entered is shown and the loop continues. If the Clear Form button is used, the form is reset and the loop continues. If the Quit button is used, the loop exits and the program ends. The end of the program uses the **rsrc_free( )** function to release the memory used by the resource file.

Function **init_form( )** simply starts the addresses and lengths of the text strings to be entered through the dialog box. The function **init_form( )** in the application functions is an example of how to access objects within a tree. Because all object trees are stored as arrays, the program needs the base address of the array. The base address of the tree is obtained through the **rsrc_gaddr( )** function. This function has three parameters. The first parameter indicates the object type to be located (see Table 10-1). The second parameter is the index number of this structure. The last parameter is an address of a pointer. The address of the located structure is placed in this parameter upon return. In FORM the type of data structure searched for is a tree. It has an index number ENTRYBOX defined in the header file for the program. The address of the tree is put into variable **box_addr**. Note: Make sure that the header file FORM.H is included in the program. If it is not included, the program does not know the index values of the object arrays.

Table 10-1: Function rsrc_gaddr( ) **Data Structure Types**

Parameter Value	*Type of Data Structure*
0	Tree
1	OBJECT
2	TEDINFO
3	ICONBLK
4	BITBLK
5	String
6	Image data
7	ob_spec
8	te_ptext
9	te_ptmplt
10	te_pvalid
11	ib_pmask
12	ib_pdata
13	ib_ptext
14	bi_pdata
15	ad_frstr—address of a pointer to a free string
16	ad_frimg—address of a pointer to a free image

Once **rsrc_gaddr( )** returns the value, **box_addr** will have the base address to an array that contains the objects in the ENTRYBOX tree. Therefore, all the objects in the ENTRYBOX tree can be accessed using the index values defined in the header file. The next eight lines initialize the form to make sure that the length and initial text are set properly. Each of the text entry fields is of type G_FTEXT. In Chapter 9, an object of this type has an OBJECT structure and a TEDINFO structure. In **init_form( )**, the variable **box_addr[LNAME]. ob_spec** refers to the **ob_spec** field of the OBJECT structure in the array for the last name entry. For a G_FTEXT object, the **ob_spec** field points to a TEDINFO structure, so that a type cast of a TEDINFO pointer is used for the **ob_spec** field. Within the TEDINFO structure, the **te_ptext** field is initialized to point to an empty string. This is an example of how to get quite confused if you don't keep the objects and their types organized. With the liberal use of object names and good comments in the program, you should be able to keep track of your program's flow.

The next function listed, **get_form( )**, provides the series of routines required to display a dialog box, get data from the user, and remove the dialog box from the screen. Displaying a dialog box through the AES follows a neat and orderly procedure. First, **rsrc_gaddr( )** is called to get the address of the object tree. Function **form_center( )** is called to center the dialog box on the screen. The dialog box does not have to be centered and can be placed anywhere you like. However, a

centered dialog box looks better. The **form_dial( )** routine is called to perform several utility functions. The first time it is called, it reserves the part of the physical screen where the dialog box is to appear. This simply prevents anything else from appearing on that portion of the screen until the dialog box is completed. A second call to **form_dial( )** draws an expanding box, much like the one displayed when a floppy disk is opened to display the directory. Function **objc_draw( )** is called to draw the dialog box and its contents. Function **form_do( )** turns control over to the AES, which then allows the user to interact with the dialog box. When the user satisfies one of the exit conditions (by pressing the Exit button or the Return key if a default button was chosen), **form_do( )** returns and **form_dial( )** are called a third time to draw a shrinking box. Function **form_dial( )** is called one more time to free the screen area and redraw the screen at that location.

In **get_form( )**, the first thing needed is the address of the dialog box. Next for demonstration purposes, the current values in the text fields are shown. These values are then cleared from the entry form. The states of the selection boxes for income level and hobbies are all set to NORMAL so that no buttons are shown as selected. It is important to initialize the values of entry fields before displaying forms such as this.

The function **form_center( )** is then called. It is given the address of the dialog box and returns the x and y coordinates and width and height of the box. The coordinates are screen coordinates measured in pixels. Once the program knows where the dialog box is located, the first call to **form_dial( )** can be made. The first parameter for **form_dial( )** determines what type of action is to be performed (see Table 10-2). The values **smallx**, **smally**, **smallw**, and **smallh** are not used in this call. They are provided here as placeholders and are used in other calls to **form_dial( )**. The variables **xbox**, **ybox**, **wbox**, and **hbox** determine the size and location of the dialog box. These values are used to reserve screen space for the dialog box.

Table 10-2: Function form_dial( ) Action Flags

Constant Name	*Value*	*Action*
FMD_START	0	Reserve screen space
FMD_GROW	1	Draw expanding box
FMD_SHRINK	2	Draw shrinking box
FMD_FINISH	3	Release screen space

The next call to **form_dial( )** actually draws the expanding box using the defined value FMD_GROW as the first parameter. The variables **smallx**, **smally**, **smallw**, and **smallh** contain the size and location of

the starting box size. Function **form_dial( )** draws the image of a box expanding from its small size and location to its large size and location. The size and location of the small and large boxes are completely arbitrary, and you may use any values that are appropriate to your program. In FORM, the small box is located at the center of the screen with no height or width.

The next step is to draw the dialog box on the screen using **objc_draw( )**. Given a tree address and the index to the first object to draw (the first two parameters), **objc_draw( )** will draw the object and any number of levels of children (the third parameter) for that object. The last four parameters indicate the clipping rectangle to use. The clipping rectangle in this case is the same size as the dialog box. An object's *level* in a tree is the distance from the root to the object. For example, the root itself is at level 0. The children of the root are at level 1. The children of those objects are at level 2, and so forth. Given a starting index value (here the start is the root, but it can be anywhere), **objc_draw( )** draws the specified number of levels from that point on. The value 10, used in FORM, is a relatively large value and should cover most dialog boxes created. However, any value can be used. A value of 32 or 64 should be sufficient to cover any tree you would create. Function **objc_draw( )** is quite flexible in what it draws. By passing the address of a tree, it can draw the entire tree, the first few levels, only one object, or a few objects in the middle of the tree.

Once the dialog box is displayed on the screen, the program is ready to accept data from the user. The AES function **form_do( )** causes the AES to take control of the system and handles user interaction with the dialog box. The AES controls what can be changed, edited, or selected based upon the flags and states for each object. When the user selects the Exit button, function **form_do( )** ends. The index of the exit object selected is the value returned by **form_do( )**. If the user pressed the Return key to exit, the index of the default exit object is returned. In **get_form( )**, the index of the exit object is saved in variable **exit_object**.

When the user has completed entering the information in the dialog box, the application should remove the box from the screen. In this case, **form_dial( )** is called to draw a shrinking box and then called again to release the area on the screen used by the dialog box.

The final action performed in routine **get_form( )** is to reset the state of the exit object. When function **form_do( )** returns control to the program, all the object states, flags, and data are left unchanged to allow the program to examine the exit condition and state of the dialog box. When a user selects an exit object, that object is put into its selected state (reverse colors). The next time the dialog box is displayed, this exit object appears selected. To avoid this confusion, the exit object must be set back to its normal state. In **get_form( )**,

once the dialog box is removed from the screen, the state of the exit object is restored to NORMAL. After resetting the state of the exit object, **get_form( )** returns the index of the exit object to the calling function.

Function **show_info( )** displays the values selected by the user. Primarily, the function demonstrates accessing the various fields of the objects in the dialog box and how to interpret this information.

Program FORM is relatively simple compared to some of the programs presented later. It is important to understand the field access method and how an index is used to access a particular object. All these features are used in the remaining programs. Experiment with program FORM. First investigate the fields of the OBJECT and TEDINFO structures. Then change the template, validation, and text strings through FORM and through the resource editor. Finally, try changing the objects in the dialog box or even creating dialog boxes of your own. FORM is a sample program to familiarize you about how objects are related between the program and the resource file.

AES Naming Conventions

There is a certain pattern in the AES function names. The AES has a large number of functions that are divided into *libraries*, or groupings of functions that have something in common. The *applications manager* deals with communications between applications and with initializing and exiting applications. All applications manager routines begin with "**appl_**". The *event manager*, or event library, contains functions to handle events that occur during the processing of a program. Events control the flow of all programs. Event manager routines begin with "**evnt_**". The *file selector manager* provides a file selector dialog box and controls its activities. The file selector manager consists of one routine, **fsel_input( )** discussed at the end of this chapter. The *form manager* controls forms: dialog boxes, alert boxes, and error boxes. All of its routines begin with "**form_**". The *graphics manager* handles basic graphic output used by the AES. It contains the actual routines that draw growing and shrinking boxes. It also allows the application to make changes in how the mouse cursor is displayed. All these routines begin with "**graf_**". The *menu manager* controls the menu bar, the menu item appearance, and registers desk accessories in the menu. Its routines begin with "**menu_**". The *object manager* allows an application to find objects, draw objects, change and reorder an object tree, and add or delete objects. All of its routines begin with "**objc_**". The *resource manager* handles loading and freeing of resource space in memory and deter-

mines addresses of resource objects. All of its routines begin with "**rsrc_**". The *scrap manager* is used to write information out to a scrap file. It contains two functions: **scrp_read( )** and **scrp_write( )**. Finally, the *window manager* handles the creation, deletion, opening, and closing of windows, as well as window maintenance. Its routines begin with "**wind_**".

Using Menus

The next two programs use a resource file called MENU.RSC. This resource file contains two trees, a menu, and a dialog box. A menu is also based on an object tree. The menu bar is the root. Each menu title is a child of the root. The entries in each menu are children of the menu title.

Load the resource editor program, and drag a menu icon into your work area. Name this tree MAINMENU. Double-click this menu icon to open it. As mentioned in Chapter 9, the menu bar should always have the two titles Desk and File. Click the Desk option to display this menu. On the first line, the option is labeled "Your message here." The second entry is a line of disabled dashes. The next six entries list "Desk Accessory 1" through "Desk Accessory 6." The six desk accessory entries are used by the AES menu manager. When a desk accessory is in the system, the menu manager changes one of the entries in the Desk menu to the title of the accessory. This is done without any action from your program. The first entry in the Desk menu is usually changed to read "About Program..." This entry is used to initiate a dialog box that provides information about the application. Under the File title, there is only one option called Quit. The menu part box should contain at least four items: a title object, an entry object, a disabled dash, and a box.

For the MENU.RSC file, you need to change some of the existing menu entries and create a new menu (see Figure 10-1). To create a

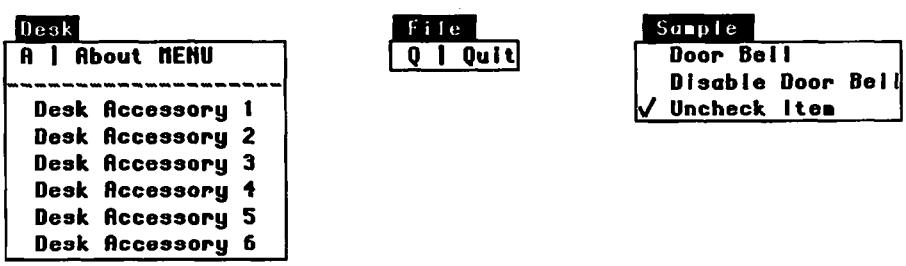


Figure 10-1 Menu Bar from MENU.RSC

new menu, drag a TITLE object from the parts box to the menu bar and place it to the right of the File title. Double-click it, and change the text to read "Sample." Make sure you include the space before the word so that it doesn't run into the File title. Click OK and the new menu title is displayed.

Click the Sample title once to select it. The title is displayed in reverse colors and has an empty entry box underneath it. This box holds all entries for the menu, so it must be long enough and wide enough for these entries. Resize the Sample entry box so that it contains about three lines. The exact size is not that important right now since you can resize the box again later.

To make an entry in a menu, drag an ENTRY object from the parts box to the first line of the menu's entry box. The left edge of the ENTRY object should be just inside the left edge of the entry box. Repeat this procedure two more times on the next two lines for the Sample menu. You now have a menu with three titles called Desk, File, and Sample. Under the Sample menu there is an entry box with its first three lines reading "ENTRY."

To change the text of an entry, double-click it, and change the text line. Double-click the top entry, and label it "Door Bell." For aesthetic purposes, most menu entries have at least two spaces before the text. The first space is used to hold a check mark (if necessary), and the second space is used to allow room between the check mark and the text. Change the text of the second entry to "Disable Door Bell," and change the text of the third entry to "Uncheck Item." For the third entry, also select the CHECKED flag to turn on the check mark.

Return to the Desk menu. Change the first entry line to read " A | About MENU". The letter "A" at the start of the entry signals the user that the Control-A keyboard entry performs the same function as selecting this item from the Desk menu. Go to the File menu and change the Quit option to read " Q | Quit." The user is able to press Control-Q instead of clicking the Quit option. You now have the completed menu (see Figure 10-1).

Close the menu to return to the main window. Drag a dialog box to the work area and call it INFOBOX. This is the dialog box displayed in response to the About MENU entry in the Desk menu. Create the text using STRING objects as shown in Figure 10-2. Add a button with the OK text. Make sure the button is set as SELECTABLE, DEFAULT, and EXIT.

The final step in creating this resource file is to name the objects used in the program. The only objects of concern in this file are in the menu. Return to the menu window and select the About MENU entry in the Desk menu. Name this object INFO. Go to the File menu, select QUIT, and name it QUIT. Select the title for the Desk menu and name it DESK. Name the File menu title FILE. The Sample menu title

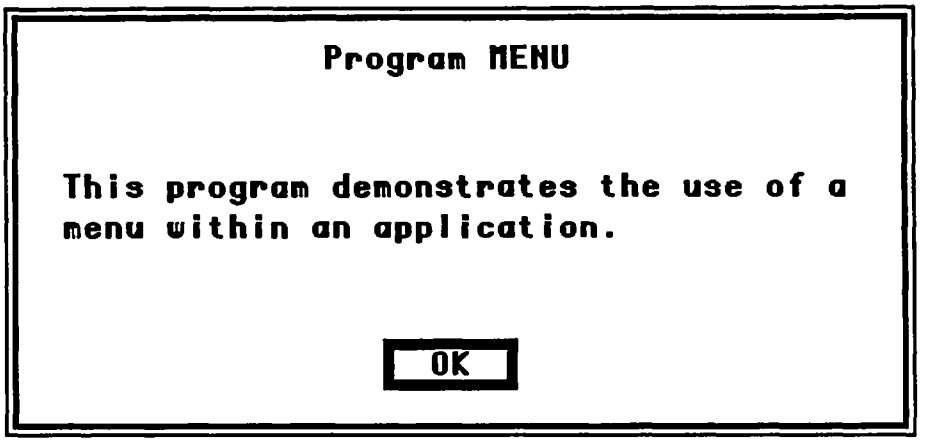


Figure 10-2 Dialog Box INFOBOX from MENU.RSC

is to be named SAMPLE. Finally, name the Door Bell entry as SAMPLE1, the Disable Door Bell entry as SAMPLE2, and the Uncheck Item as SAMPLE3.

This completes the resource file for programs MENU1 and MENU2. Save this resource file in the file called MENU.RSC.

Program MENU1

Program MENU1 provides an introduction to using a menu in your program. The menu bar is one portion of the screen image handled by the AES. A program can request a new menu bar to be displayed and can change how the items in the menu are displayed. However, any user interaction with the menu is handled by the AES. Whenever the mouse touches one of the menu titles, the AES immediately takes control and displays the drop-down menu. The AES retains control of the system until the mouse button is pressed. Pressing the mouse button can signal a menu selection if the mouse is located on an active menu item. Once the mouse button is pressed the screen image beneath the drop-down menu is restored. If a menu item is selected, the AES communicates the selection to the application by sending it a message as described in Chapter 9.

Receiving a message is an event. To receive the message, the program has to wait for a message event to occur. Only when a message event is received should the program try to read a message. Waiting for a message event is done through the **evnt_mesag( )** function from the event manager. This function has one parameter—a pointer to an array of eight WORDs. In essence, this array provides a

16-byte buffer for the message. The message buffer has a predefined format depending upon the type of message sent. Element 0 of the array contains the type of message sent. Element 1 is the ID number of the application that originates the message. Element 2 contains the message length in excess of 16 bytes. If this value is 0, the message is less than or equal to the basic 16 bytes. If the message is larger than 16 bytes, the value in element 2 is equal to the message length minus 16. In any case, the first three elements of the 16-byte message buffer are always used. Any excess bytes must be read using the **appl_read( )** function described in Appendix A. The meaning of the remainder of the message buffer depends upon the type of message sent (see Appendix B for description of messages).

Program MENU1 is only interested in one type of message—a menu-selected message. The predefined constant MN_SELECTED is the message type used to indicate that a menu item has been selected by the user. For this type of message, element 3 of the message buffer array contains the index number of the menu title object. Element 4 contains the index number of the menu item selected. For example, if the user selects Disable Door Bell, element 3 has the index number of title Sample, and element 4 has the index number of the entry Disable Door Bell.

Look at program MENU1 in Listing 10-2. In function **main( )**, the initialization is the same as in program FORM except that the resource file loaded is MENU.RSC. The address of the menu tree is retrieved and function **menu_bar( )** is called to display the new menu. Function **menu_bar( )** has two parameters. The first is the address of the menu. The second is whether the menu is to be displayed. In some cases, you may want to make the menu bar invisible until the user performs some action like changing disks.

Listing 10-2 Program MENU

```
/*********************************************************************
   MENU1.C Menu demonstration program

   This program shows the use of menus in an application.
*********************************************************************/

/*************************************************
   System Header Files & Constants
*************************************************/

#include <stdio.h>                /* Standard IO */
#include <osbind.h>               /* GEMDOS routines */
#include <gemdefs.h>              /* GEM AES */
#include <obdefs.h>               /* GEM constants */
```

Listing 10-2 (continued)

```
#define  FALSE   0
#define  TRUE    !FALSE

/*************************************************
   GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int WORD;                  /* WORD is 16 bits */
WORD      contrl[12],                /* VDI control array */
          intout[128], intin[128],   /* VDI input arrays */
          ptsin[128], ptsout[128];   /* VDI output arrays */

WORD      screen_vhandle,            /* virtual screen workstation */
          screen_phandle,            /* physical screen workstation */
          screen_rez,                /* screen resolution 0,1, or 2 */
          color_screen,              /* flag if color monitor */
          x_max,                     /* max x screen coord */
          y_max;                     /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

#include  'menu.h'

char ding_dong[] = {                 /* array for Dosound */
     0, 239, 1, 0, 7, 62, 8, 8, 130, 25,
     0, 63, 1, 1, 130, 40, 7, 63, 8, 0, 255, 0
};

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;
```

Listing 10-2 (continued)

```
for (i = 0; i < 10; i++)                /* set      for default values */
          work_in[i] = 1;
     work_in[10] = 2;                   /* use raster coords */
     new_handle = phys_handle;          /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();         /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

/*************************************************
     Application Functions
*************************************************/

do_dialog(box_index)
WORD box_index;
/*************************************************
Function: Display a dialog box.
Input:    box_addr  = index of dialog box
Output:   Returns index of object used for exit.
*************************************************/
{
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of box */
     rsrc_gaddr(0, box_index, &box_addr);

/* get size and location of a box centered on screen */
     form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
     smallx = xbox + (wbox / 2);
     smally = ybox + (hbox / 2);
     smallw = 0;
     smallh = 0;
```

Listing 10-2 (continued)

```
/* reserve area on screen for box display */
     form_dial(FMD_START,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw an expanding box */
     form_dial(FMD_GROW,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw dialog box */
     objc_draw(box_addr, 0, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
     exit_object = form_do(box_addr, 0);

/* draw a shrinking box */
     form_dial(FMD_SHRINK,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
     form_dial(FMD_FINISH,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reset exit object state to unselected */
     box_addr[exit_object].ob_state = NORMAL;

     return(exit_object);
}


/***************************************************
     Main Program
***************************************************/


main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/* new data variables */
OBJECT    *menu_addr;               /* address for menu */
WORD      msg_buf[8];               /* message buffer */
```

Listing 10-2 (continued)

```
/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();          /* Initialize AES routines */
     if (ap_id < 0)                /* no calls can be made to AES */
     {                             /* use GEMDOS */
          Cconws("***) Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                /* set exit value to show error */
     }

     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */


/*************************************************
     Application Specific Routines
*************************************************/

     if (!rsrc_load("MENU.RSC"))
     {
          form_alert(1, "[0][Cannot file MENU.RSC file|Exiting ...][OK]");
          exit(1);
          exit(1);
     }

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);

/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* wait for a message indicating a menu selection */
     for(;;)                       /* continue loop until quit */
     {
          evnt_mesag(msg_buf);
          if (msg_buf[0] != MN_SELECTED)/* not a menu message */
               continue;                /* then ignore */
          if (msg_buf[4] == QUIT)
               break;              /* exit loop */
          switch(msg_buf[4])       /* find object index */
          {
          case SAMPLE1:            /* ring bell */
               Dosound(ding_dong);
               break;
```

Listing 10-2 (continued)

```
        case SAMPLE2:           /* change item 1 state */
             if (menu_addr[SAMPLE1].ob_state == DISABLED)
             {
                  menu_addr[SAMPLE1].ob_state = NORMAL;
                  strcpy(menu_addr[SAMPLE2].ob_spec,
                       " Disable Door Bell");
             }
             else
             {
                  menu_addr[SAMPLE1].ob_state = DISABLED;
                  strcpy(menu_addr[SAMPLE2].ob_spec,
                       " Enable Door Bell");
             }
             break;

        case  SAMPLE3:          /* check or uncheck item */
             if (menu_addr[SAMPLE3].ob_state == CHECKED)
             {
                  menu_addr[SAMPLE3].ob_state = NORMAL;
                  strcpy(menu_addr[SAMPLE3].ob_spec,
                       " Check item");
             }
             else
             {
                  menu_addr[SAMPLE3].ob_state = CHECKED;
                  strcpy(menu_addr[SAMPLE3].ob_spec,
                       " Uncheck item");
             }
             break;

        case INFO:              /* display program info */
             do_dialog(INFOBOX);
             break;

        default:
             break;
        }
                                /* reset title state */
        menu_addr[msg_buf[3]].ob_state = NORMAL;
        menu_bar(menu_addr, TRUE);
    }                           /* end infinite loop */

/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     rsrc_free();
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
```

After displaying the new menu bar, MENU1 enters an infinite loop. From this point on, all program flow is controlled by events. When an event occurs, the program responds with the appropriate action and waits for the next event. Only when an event indicating an exit condition occurs (such as selecting the Quit option from the menu) does the program exit this infinite loop.

The first function call within the loop is a call to **evnt_mesag( )**. When the program reaches this statement during execution, it waits until any message event occurs. Any other type of event (such as a button pressed down or a mouse movement) is ignored because the program is only waiting for a message to be received. When a message is received, the message buffer is filled and the program continues. In program MENU1, if the message is not a menu selection message (that is, **msg_buf[0]** is not equal to MN_SELECTED), the program waits for another message event.

If a menu-selected message is received, element 4 of the message buffer array is checked to see which menu item is selected. If the menu item Quit is selected, the program breaks out of the loop and exits to the desktop. Otherwise, a switch statement on this message array element selects the appropriate action. If the selection is item SAMPLE1 (Door Bell) the **do_sound( )** function gives the common two-tone door bell sound.

If the item is SAMPLE2 (Disable Door Bell), the program checks the current state of the Door Bell entry. If the door bell is currently disabled, it is enabled and item SAMPLE2 is set to read "Disable Door Bell" so that it can be disabled the next time through. If the door bell is currently enabled, it is disabled and item SAMPLE2 is set to read "Enable Door Bell."

In case SAMPLE3, a similar toggle situation occurs. If the item is currently checked, the item is unchecked. If the item is currently unchecked, the item is checked. The text of the selection is changed accordingly.

In case INFO, the user is requesting information about the menu program. Function **do_dialog( )** in the application functions section of MENU1 is called with parameter INFOBOX. This function is simply a generalized version of the **get_info( )** function used in program FORM.

If the item selected does not match any of the cases, the default case ignores it. The last two statements in the loop reset the title of the selected menu item. When a menu item is selected, the AES removes the drop-down menu from the screen and leaves the menu title in its selected state (that is, reverse colors). For example, if you touch the File menu with the mouse at the desktop, the title "File" is placed in a selected mode and stays that way until the operation has been completed. When the AES reports the message to your applica-

tion, the title is already in its selected state. When your program has completed the selected operation, it must reset the menu title state back to the normal state. When the menu object state is changed, the menu bar must be redisplayed to show the title in its normal state. Element 3 in the message array contains the index to the title of the menu item selected.

MENU1 does not wait for any keyboard events so nothing happens when keys are pressed. Your program must handle the special shortcut key codes set up in the resource file such as the Control-A to get program information and Control-Q to quit. Program MENU2 adds this feature to MENU1. Look through MENU1 before continuing. Make sure that you understand the concept of receiving messages and how the elements of the message array are used. Then continue to the next program.

Program MENU2

Program MENU2 is an enhanced version of MENU1. MENU2 even uses the same resource file as MENU1. MENU2, however, handles more events. Specifically, MENU2 handles both message and keyboard events.

In function **main( )** in MENU2 (see Listing 10-3), the initialization and cleanup are the same as in MENU1. The primary enhancement over MENU1 is that the control loop for event processing has been moved out of **main( )** and into its own function called **control( )**. Function **control( )** is used throughout the rest of this book.

Listing 10-3 Program MENU2

```
/*********************************************************************
     MENU2.C Menu demonstration program (version 2)

     This program shows the use of menus in an application
     with multiple events.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */

#define   FALSE     0
#define   TRUE      !FALSE
```

Listing 10-3 (continued)

```
/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;                /* WORD is 16 bits */
WORD      contrl[12],               /* VDI control array */
          intout[128], intin[128],  /* VDI input arrays */
          ptsin[128], ptsout[128];  /* VDI output arrays */

WORD      screen_vhandle,           /* virtual screen workstation */
          screen_phandle,           /* physical screen workstation */
          screen_rez,               /* screen resolution 0,1, or 2 */
          color_screen,             /* flag if color monitor */
          x_max,                    /* max x screen coord */
          y_max;                    /* max y screen coord */

/*************************************************
     Application Specific Data
*************************************************/

#include  "menu.h"
#define   QUIT_KEY 0x1011           /* control-Q to quit */
#define   INFO_KEY 0x1e01           /* control-A for about Info */

char ding_dong[] = {                /* array for Dosound */
     0, 239, 1, 0, 7, 62, 8, 8, 130, 25,
     0, 63, 1, 1, 130, 40, 7, 63, 8, 0, 255, 0
};

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          nsw_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
```

Listing 10-3 (continued)

```
     new_handle = phys_handle;          /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}


/*************************************************
     Application Functions
*************************************************/

do_dialog(box_index)
WORD box_index;
/*************************************************
Function: Display a dialog box.
Input:    box_addr    = index of dialog box
Output:   Returns index of object used for exit.
*************************************************/
{
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of box */
     rsrc_gaddr(0, box_index, &box_addr);

/* get size and location of a box centered on screen */
     form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
     smallx = xbox + (wbox / 2);
     smally = ybox + (hbox / 2);
     smallw = 0;
     smallh = 0;

/* reserve area on screen for box display */
     form_dial(FMD_START,
```

Listing 10-3 (continued)

```
            smallx, smally, smallw, smallh,
            xbox, ybox, wbox, hbox);

/* draw an expanding box */
     form_dial(FMD_GRDW,
            smallx, smally, smallw, smallh,
            xbox, ybox, wbox, hbox);

/* draw dialog box */
     objc_draw(box_addr, 0, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
     exit_object = form_do(box_addr, 0);

/* draw a shrinking box */
     form_dial(FMD_SHRINK,
            smallx, smally, smallw, smallh,
            xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
     form_dial(FMD_FINISH,
            smallx, smally, smallw, smallh,
            xbox, ybox, wbox, hbox);

/* reset exit object state to unselected */
     box_addr[exit_object].ob_state = NORMAL;

     return(exit_object);
}

control()
/*************************************************
Function: Master control function.
Input:    None. Program initialization must be done before
          entering this function.
Output:   None. Returns for normal program termination.
*************************************************/
{
OBJECT    *menu_addr;              /* address for menu */
WORD      mevx, mevy,              /* evnt_multi parameters */
          mevbut,
          keystate,
          keycode,
          mbreturn,
          msg_buf[8];              /* message buffer */
WORD      event,                   /* evnt_multi result */
          menu_index;              /* keyboard menu title selected */

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);
```

Listing 10-3 (continued)

```
/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* wait for a message indicating a menu selection */
for(;;)                              /* continue loop until quit */
{
     event = evnt_multi( (MU_KEYBD | MU_MESAG),
          0,                         /* # mouse clicks */
          0,                         /* mouse buttons of interest */
          0,                         /* button state */
          0,                         /* first rectangle flags */
          0, 0,                      /* x,y of 1st rectangle */
          0, 0,                      /* height, widht of 1st rect */
          0,                         /* second rectand flags */
          0, 0,                      /* x,y of 2nd rect */
          0, 0,                      /* w,h of 2nd rect */
          msg_buf,                   /* message buffer */
          0, 0,                      /* low, high words for timer */
          &mevx, &mevy,              /* x,y of mouse event */
          &mevbut,                        /* button state at event */
          &keystate,                 /* status of keyboard at event */
          &keycode,                  /* keyboard code for key pressed */
          &mbreturn);                /* # times mouse key enter state */

     if (event & MU_MESAG)
     {
          if (msg_buf[0] != MN_SELECTED)      /* not a menu message */
               continue;                      /* then ignore */
          if (msg_buf[4] == QUIT)
               break;                /* exit loop */

          switch(msg_buf[4])         /* find object index */
          {
          case SAMPLE1:              /* ring bell */
               Dosound(ding_dong);
               break;

          case SAMPLE2:              /* change item 1 state */
               if (menu_addr[SAMPLE1].ob_state == DISABLED)
               {
                    menu_ienable(menu_addr, SAMPLE1, 1);
                    menu_text(menu_addr, SAMPLE2,
                         "  Disable Door Bell");
               }
               else
               {
                    menu_ienable(menu_addr, SAMPLE1, 0);
                    menu_text(menu_addr, SAMPLE2,
                         "  Enable Door Bell");
               }
               break;
```

Listing 10-3 (continued)

```
            case  SAMPLE3:              /* check or uncheck item */
                 if (menu_addr[SAMPLE3].ob_state == CHECKED)
                 {
                      menu_icheck(menu_addr, SAMPLE3, 0);
                      menu_text(menu_addr, SAMPLE3, '  Check item');
                 }
                 else
                 {
                      menu_icheck(menu_addr, SAMPLE3, 1);
                      menu_text(menu_addr, SAMPLE3, '  Uncheck item');
                 }
                 break;
            case INFO:                  /* display program info */
                 do_dialog(INFOBOX);
                 break;


            default:
                 break;
            }
                                        /* reset title state */
            menu_tnormal(menu_addr, msg_buf[3], 1);
       }                                /* end message handler */


       if (event & MU_KEYBD)
       {
            if (keycode == QUIT_KEY)
            {
                 menu_index = FILE;
                 menu_tnormal(menu_addr, menu_index, 0);
                 break;
            }


            switch(keycode)
            {
            case INFO_KEY:
                 menu_index = DESK;
                 menu_tnormal(menu_addr, menu_index, 0);
                 do_dialog(INFOBOX);
                 break;


            default:
                 break;
            }
            menu_tnormal(menu_addr, menu_index, 1);
       }                                /* end keyboard handler */
  }                                     /* end infinite loop */
       return;
  }                                     /* end function */
```

Listing 10-3 (continued)

```
/*************************************************
     Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
     {                              /* use GEMDOS */
          Cconws('***> Initialization Error. <***\n');
          Cconws('Press any key to continue.\n');
          Crawcin();
          exit(-1);                 /* set exit value to show error */
     }

     screen_phandle =               /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();             /* Get screen attributes */
/*************************************************
     Application Specific Routines
*************************************************/

     if (!rsrc_load("MENU.RSC"))
     {
          form_alert(1, "[0][Cannot file MENU.RSC file|Exiting ...][OK]");
          exit(1);
     }

control();

/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     rsrc_free();
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
/*************************************************/
```

Function **control( )** is basically the same set of procedures used in MENU1. First, the address of the menu is obtained and the menu is displayed. Then an infinite loop is used for event processing. The first statement in the loop is a call to **event_multi( )** rather than **event_mesg( )**. Function **event_multi( )** waits for a specified event or combination of events. There are button events, keyboard events, message events, mouse events, and timer events. Button events indicate that one or more of the mouse buttons have been pressed. Keyboard events indicate that a key on the keyboard has been pressed. Message events are sent from the AES to the application or from one application to another. Mouse events occur when the mouse enters or leaves a particular area on the screen. A rectangle is used as a parameter to specify the sensitive area. Timer events let a certain amount of time pass before an event happens.

It is not possible to use individual event functions when a program must be ready to accept any of a set of events to occur. For example, a program must wait for a button, a keypress, or a message event. These events may occur in any order. If the program uses an **evnt_button( )** function to wait for the mouse button, followed by an **evnt_keybd( )** function to accept the keypress, and then an **evnt_mesag( )** function, only one event is accepted at a time. When this program executes, it reaches the **evnt_button( )** function and waits for a mouse button to be pressed. Regardless of the number of keypresses or messages that occur, program execution ceases until a mouse button is pressed. To avoid this single event situation, a program must use the **evnt_multi( )** function.

The **evnt_multi( )** function waits for any of a set of events. By specifying the types of events to wait for, a program can accept the first event that comes along and process it. Take a look at the **evnt_multi( )** call in function **control( )**. Note that it has quite a number of parameters: 23 to be precise. The first parameter is a flag indicating which types of events to wait for and accept. The event types are defined as constants in the GEMDEFS.H file (see Table 10-3). Each event type is associated with a different bit in this

Table 10-3: Event Type for Function evnt_multi( )

Constant Name	*Value*	*Event*
MU_KEYBD	0X01	Keyboard
MU_BUTTON	0X02	Mouse button
MU_M1	0X04	Mouse event 1
MU_M2	0X08	Mouse event 2
MU_MESAG	0X10	Message
MU_TIMER	0X20	Timer

parameter. By ORing the event types together you can wait for any combination of events.

The next parameter in the **evnt_multi( )** function is the number of mouse clicks that cause a button event. The value of this parameter determines the number of clicks necessary to initiate the event. The time interval allowed between the clicks is determined by the control panel accessory. The third parameter lists the mouse buttons of interest. The least significant bit (bit 0) corresponds to the furthest left mouse button. The next bit corresponds to the next button to the right, and so on. If a bit for a particular button is not set, that button is ignored. The fourth parameter is the button state. It is constructed in the same manner as the previous parameter. If the bit is set to 0 for the button state, the event is looking for an "up" condition on the button. If the bit is set to 1, the event looks for a "down" condition on the button.

A mouse event occurs when the mouse cursor enters or leaves a specified area on the screen. Two different areas may be listed, corresponding to a mouse event 1 or a mouse event 2. The fifth parameter of **evnt_multi( )** is a flag for the first rectangle. If this value is 0, the mouse event 1 is generated upon entry of the mouse into the rectangle. If the value is 1, the event occurs upon exit of the rectangle. The next two parameters give the x and y values for the upper left corner of the first rectangle (in screen coordinates). The following two parameters are the height and width (also in screen coordinates) of the first rectangle. The next five parameters are the flag, the x and y coordinates, and the width and height of the second rectangle.

The next parameter in **evnt_multi( )** is a pointer to the message buffer array. In the case of a message event, this buffer is filled with the message.

The next two parameters contain the number of milliseconds to wait for a timer event. The number of milliseconds is essentially a **long** value. A **long** value can be divided into two words: a high word and a low word. The first parameter listed for **evnt_multi( )** contains the low word value and the second parameter the high word value.

The next two parameters hold the returned values of the x and y coordinates (in screen coordinates) of the mouse when a mouse event has occurred. The next parameter holds the button states at the time of a button event. This value has the same format as the button-state parameter mentioned above.

The parameter **keystate** holds the status of the keyboard, specifically the right shift, left shift, Control, and Alternate keys. These correspond to bits 0 through 3, respectively. The bits are set to 1 if the particular key is depressed at the time of the event. The **keycode** variable returns the keyboard code for the key pressed in a keyboard

event. The keycodes are listed in Appendix C. The last parameter holds a value returned. This value indicates the number of times the mouse button was clicked.

The **evnt_multi( )** function also returns a value. This value corresponds to the event that occurs. In program MENU2, this value is stored in variable **event**.

When **evnt_multi( )** is called, the application waits for the specified events to occur. MENU2 waits for a keyboard event and a message event. The message event is needed because the user can select a menu item. The keyboard event is used to collect any possible keyboard shortcut commands.

If a message event occurs, processing proceeds exactly as in MENU1. However, this program uses menu manager functions to set the text and state of the menu selections; for example, **menu_ienable( )** is used to enable a menu item and **menu_text( )** is used to set the text of an item. Function **menu_icheck( )** is used in case SAMPLE3 is used to set the checked status of the menu item. At the end of the switch statement, the function **menu_tnormal( )** is used to reset the menu title to its normal state. Note that the menu bar does not have to be redrawn in this case.

If a keyboard event occurs, the program checks if the Quit key is selected. If not, the program checks if the Info key is selected. Even though there is only one condition to accept, a switch statement is used in **control( )** to make it a more generalized control function. In a keyboard event that provides for quick menu access, the menu title is not highlighted. To remain consistent with the operation of the GEM user interface, the program should highlight the menu title of the item selected. Use **menu_tnormal( )** to set the highlight. The variable **menu_index** keeps track of which menu title index is highlighted. When event processing is completed, the menu title can be reset to its normal state.

MENU2 demonstrates how to use the **evnt_multi( )** function as well as some menu manager functions. Work with MENU2, and add some more keyboard shortcuts. At this point, you are ready to create a basic GEM application.

Program LISTER

Now that you have created programs that use forms, menus, and events, you are ready to write a small application program. Some of the points you should look at in program LISTER are the consistency of the user interface, the organization of the **control( )** function, the modularity of design, and the use of the file selector function, **fsel_input( )**.

Program LISTER is a program that displays the contents of a disk file. Most programmers use two different formats for the file display. One format simply lists the contents of the file on the screen. This format is useful for files that contain text only (characters with ASCII values from 32 through 127). For other types of files such as program or resource files, this format would be meaningless because the display would be a series of random characters. In this case, the second display format, called a *dump* format, is used. A dump of a file consists of a sequence of lines divided into two parts. Each line displays 16 bytes of the file. On the left side of the line is the hexadecimal representation of the byte values. On the right side of the line is the character (also called graphics) representation of each.

Program LISTER allows the user to display a file in either format. To operate this program, the user selects one display format from the menu. The program displays the file selector dialog box provided by the **fsel_input()** function so that the user can choose the file to be displayed. LISTER then displays the selected file in the appropriate format. If the file display spans across more than one screen, the program pauses and waits for the user to press a key. When the file display finishes, the program returns to the menu bar.

The resource file for LISTER is similar to the resource file used for programs MENU1 and MENU2. The LISTER resource file contains one menu and one dialog box. Figure 10-3 shows the menu bar for program LISTER. The menu icon is called MAINMENU. Create this menu as shown in the figure. Name the About LISTER selection in the Desk menu desk object INFO. For the File menu, give the title File the name FILE, label the Text Display selection TEXTFILE, label the Dump File selection DUMPFILE, and label the Quit selection QUIT.

The dialog box for LISTER is shown in Figure 10-4. This box simply contains a brief information statement about the program. The box is named INFOBOX.

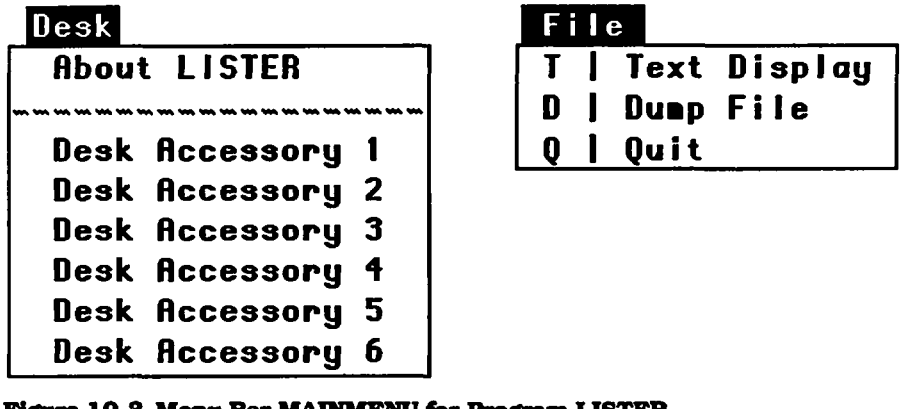


Figure 10-3 Menu Bar MAINMENU for Program LISTER

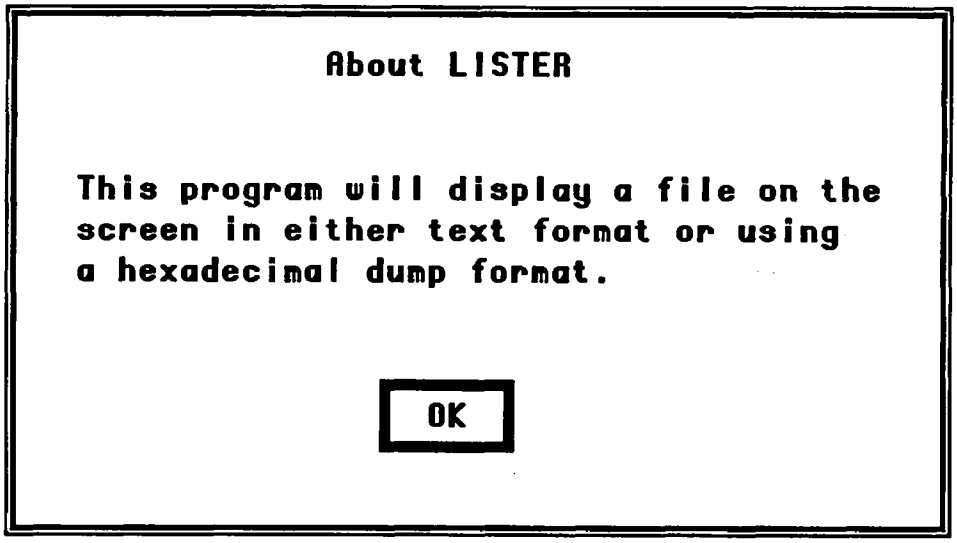


Figure 10-4 Dialog Box INFOBOX for Program LISTER

Program LISTER has a new GEM application overhead variable (see Listing 10-4). The variable **m_hidden**, initially set to FALSE, keeps track of the current visibility status of the mouse cursor. The mouse cursor can be either visible (shown on the screen) or invisible. To change the status of the mouse cursor, two new functions have been added to the GEM-related functions: **hide_mouse( )** and **show_mouse( )**. In **hide_mouse( )**, if **m_hidden** is FALSE (meaning the mouse cursor is currently visible), a call is made to the AES **graf_mouse( )** function to hide the mouse cursor and **m_hidden** is set accordingly. Function **show_mouse( )** performs the opposite task to make the mouse cursor visible. It is possible to call the **graf_mouse( )** function directly to hide and show the mouse cursor. However, the **graf_mouse( )** function can be *nested*. If you make three consecutive calls to **graf_mouse( )** to make the mouse cursor invisible, you need three consecutive calls to **graf_mouse( )** to make the mouse cursor visible again. Trying to keep the calls to **graf_mouse( )** balanced can be confusing in a large program. To avoid this problem, the functions **hide_mouse( )** and **show_mouse( )** and the global variable **m_hidden** have been added to the GEM sections of the program. If the mouse cursor is already hidden, a call to **hide_mouse( )** does nothing. If the mouse cursor is already visible, a call to **show_mouse( )** does nothing. Only if the mouse cursor is in the opposite state do the functions change the visibility state. Further discussion of the **graf_mouse( )** function is presented in Chapter 11.

Listing 10-4 Program LISTER

```
/*********************************************************************
     LISTER.C File display program

     This program will display a disk file in text or hexadecimal
     format.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard ID */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */
#include  <errno.h>                /* errno declaration */

#define   FALSE     0
#define   TRUE      !FALSE

/* The Megamax compiler was giving an error when the FILE
     type definition was used. To avoid this problem, the
     FILE type was redefined as type FP. This error occurred
     in this program ONLY.
*/
typedef FILE FP;

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;               /* WORD is 16 bits */
WORD      contrl[12],              /* VDI control array */
          intout[128], intin[128], /* VDI input arrays */
          ptsin[128], ptsout[128]; /* VDI output arrays */

WORD      screen_vhandle,          /* virtual screen workstation */
          screen_phandle,          /* physical screen workstation */
          screen_rez,              /* screen resolution 0,1, or 2 */
          color_screen,            /* flag if color monitor */
          x_max,                   /* max x screen coord */
          y_max,                   /* max y screen coord */
          m_hidden = FALSE;        /* mouse visibility status */

/*************************************************
     Application Specific Data
*************************************************/

#include  'lister.h'
#define   QUIT_KEY 0x1011          /* control-Q to quit */
```

Listing 10-4 (continued)

```
#define   DUMP_KEY 0x2004          /* control-D for dump file */
#define   TEXT_KEY 0x1414          /* control-T for text file */

#define   LINE_FEED 0x0a
#define   ESCAPE 27
#define   CR  13

int max_lines = 23;                /* number of lines to print
                                        before pausing */
char def_search[32] =              /* default search path */
          "A:\*.*",
     sel_file[16],                 /* file selected */
     file_name[64];                /* full file name to open */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
```

Listing 10-4 (continued)

```
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}

hide_mouse()
/*************************************************
Function: Make mouse invisible if currently visible.
Input:    None. Uses variable m_hidden.
Output:   Sets m_hidden to TRUE.
*************************************************/
{
     if (!m_hidden)
     {
          graf_mouse(M_OFF, 0x0L);
          m_hidden = TRUE;
     }
}

show_mouse()
/*************************************************
Function: Make mouse visible if currently invisible.
Input:    None. Uses m_hidden.
Output:   None. Sets m_hidden to FALSE.
*************************************************/
{
     if (m_hidden)
     {
          graf_mouse(M_ON, 0x0L);
          m_hidden = FALSE;
     }
}

/*************************************************
     Application Functions
*************************************************/

do_dialog(box_index)
WORD box_index;
/*************************************************
Function: Display a dialog box.
Input:    box_addr  = index of dialog box
Output:   Returns index of object used for exit.
*************************************************/
{
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of box */
     rsrc_gaddr(0, box_index, &box_addr);
```

Listing 10-4 (continued)

```
/* get size and location of a box centered on screen */
     form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
     smallx = xbox + (wbox / 2);
     smally = ybox + (hbox / 2);
     smallw = 0;
     smallh = 0;

/* reserve area on screen for box display */
     form_dial(FMD_START,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw an expanding box */
     form_dial(FMD_GROW,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw dialog box */
     objc_draw(box_addr, 0, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
     exit_object = form_do(box_addr, 0);

/* draw a shrinking box */
     form_dial(FMD_SHRINK,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
     form_dial(FMD_FINISH,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reset exit object state to unselected */
     box_addr[exit_object].ob_state = NORMAL;

     return(exit_object);
}

get_file()
/*************************************************
Function: Get a file name using file selector.
Input:    None. Uses def_search[] and sel_file[].
Output:   FALSE if cancelled, TRUE otherwise.
          Sets default file search, file selected,
          and file_name[] to contain the full file name.
*************************************************/
{
WORD exit_button;
char *temp;
int  i;
```

Listing 10-4 (continued)

```
     fsel_input(def_search, sel_file, &exit_button);
     if (!exit_button)                  /* cancelled */
          return(exit_button);

     strcpy(file_name, def_search);     /* set path */
     i = strlen(file_name);
     temp = file_name + (i - 1);        /* point to last character */
     while ( (*temp != '\\')            /* search for path end */
               && (*temp != ':')        /* drive id */
               && (temp >= file_name) ) /* or start of string */
          temp--;
     temp++;                            /* move to next position */
     strcpy(temp, sel_file);            /* add file name */
     return(exit_button);
}

FP *open_file(xx)
int xx;
/**************************************************
Function: Open file specified in file_name[] for read only.
Input:    None. Uses file_name[].
Output:   Returns file descriptor or NULL.
**************************************************/
{
FP *file_des;

     file_des = fopen(file_name, "br");
     if (file_des == NULL)
          form_error(-errno);       /* errors are negative values */
     return(file_des);
}

end_page()
/**************************************************
Function: Handle end of page condition.
Input:    None.
Output:   FALSE if abort requested.
          TRUE otherwise.
**************************************************/
{

     Cconws("Press any key to continue or ESC to abort: ");
     if ( (Crawcin() & 0x7f) == ESCAPE)
          return(FALSE);
     Cconout(CR);                   /* return to start of line */
     v_eeol(screen_vhandle);        /* erase to end of line */
     return(TRUE);
}

text_display()
/**************************************************
Function: Display a file in text format.
```

Listing 10-4 (continued)

```
Input:    None.
Output:   None.
*************************************************/
{
int  line_num,                    /* # of lines printed */
     in_char,                     /* char from file */
     i;
FP   *file_num;                   /* file descriptor */

/* have user select a file */
     if (!get_file())             /* file selection canceled */
          return;

/* open file for reading */
     file_num = open_file();
     if (file_num == NULL)        /* error opening file */
          return;

/* initialize for reading */
     hide_mouse();
     v_clrwk(screen_vhandle);
     v_curhome(screen_vhandle);

     line_num = 0;
     while ((in_char = getc(file_num)) != EOF)
     {
          printf("%c", in_char);
          if (in_char == LINE_FEED)
          {                       /* next line */
               printf("\r");      /* insure at start of new line */
               line_num++;
          }
          if (line_num == max_lines)
          {                       /* end of page */
               if (!end_page())   /* listing aborted */
                    break;
               else
                    line_num = 0;
          }
     }                            /* end while loop */
     fclose(file_num);
     Cconws("END OF FILE");
     Crawcin();
     v_clrwk(screen_vhandle);
     show_mouse();
}

dump_display()
/*************************************************
Function: Display a file in hexadecimal format.
Input:    None.
```

Listing 10-4 (continued)

```
Output:   None.
************************************************/
{
int  line_num,                    /* # of lines printed */
     count,                       /* number of chars read */
     i;
FP   *file_num;                   /* file descriptor */
char in_char[16],                 /* char from file */
     temp[80];                    /* temporary format string */

/* have user select a file */
     if (!get_file())             /* file selection canceled */
          return;

/* open file for reading */
     file_num = open_file();
     if (file_num == NULL)        /* error opening file */
          return;

/* initialize for reading */
     hide_mouse();
     v_clrwk(screen_vhandle);
     v_curhome(screen_vhandle);

     line_num = 0;
     while ((count = fread(in_char, sizeof(char), 16, file_num)) != 0)
     {
          for (i = 0; i < count; i++)   /* format hexadecimal area */
               sprintf(&temp[i*3], " %02x", in_char[i]);
          temp[count*3] = 0;            /* put null at end */
          printf("%-60s", temp);        /* output left justified */
          for (i = 0; i < count; i++)   /* output character area */
               if (in_char[i] < ' ' || in_char[i] > '~')
                    printf(".");        /* non-printable char */
               else
                    printf("%c", in_char[i]);
               printf("\n");            /* next line */
               line_num++;
          if (line_num == max_lines)    /* end of page */
               if (!end_page())
                    break;              /* listing aborted */
               else
                    line_num = 0;
     }                            /* end while loop */
     fclose(file_num);
     Cconws("END OF FILE");
     Crawcin();
     v_clrwk(screen_vhandle);
     show_mouse();
}
```

Listing 10-4 (continued)

```
control()
/*************************************************
Function: Master control function.
Input:    None. Program initialization must be done before
          entering this function.
Output:   None. Returns for normal program termination.
*************************************************/
{
OBJECT    *menu_addr;              /* address for menu */
WORD      mevx, mevy,              /* evnt_multi parameters */
          mevbut,
          keystate,
          keycods,
          mbreturn,
          msg_buf[8];              /* message buffer */
WORD      event,                   /* evnt_multi result */
          menu_index;              /* keyboard menu title selected */

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);

/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* wait for a message indicating a menu selection */
for(;;)                            /* continue loop until quit */
{
     event = evnt_multi( (MU_KEYBD | MU_MESAG),
          0,                       /* # mouse clicks */
          0,                       /* mouse buttons of interest */
          0,                       /* button state */
          0,                       /* first rectangle flags */
          0, 0,                    /* x,y of 1st rectangle */
          0, 0,                    /* height, widht of 1st rect */
          0,                       /* second rectand flags */
          0, 0,                    /* x,y of 2nd rect */
          0, 0,                    /* w,h of 2nd rect */
          msg_buf,                 /* message buffer */
          0, 0,                    /* low, high words for timer */
          &mevx, &mevy,            /* x,y of mouse event */
          &mevbut,                      /* button state at event */
          &keystate,               /* status of keyboard at event */
          &keycode,                /* keyboard code for key pressed */
          &mbreturn);              /* # times mouse key enter state */

     if (event & MU_MESAG)
     {
          if (msg_buf[0] != MN_SELECTED)      /* not a menu message */
               continue;                      /* then ignore */
          if (msg_buf[4] == QUIT)
               break;              /* exit loop */
```

Listing 10-4 (continued)

```
        switch(msg_buf[4])          /* find object index */
        {
        case TEXTFILE:              /* display file as text */
             text_display();
             menu_bar(menu_addr, FALSE);
             break;

        case DUMPFILE:              /* display file in dump */
             dump_display();
             menu_bar(menu_addr, FALSE);
             break;

        case INFO:                  /* display program info */
             do_dialog(INFOBOX);
             break;

        default:
             break;
        }
                                    /* reset title state */
        menu_tnormal(menu_addr, msg_buf[3], 1);
        menu_bar(menu_addr, TRUE);
   }                                /* end message handler */

   if (event & MU_KEYBD)
   {
        if (keycode == QUIT_KEY)
        {
             menu_index = FILE;
             menu_tnormal(menu_addr, menu_index, 0);
             break;
        }

        switch(keycode)
        {
        case TEXT_KEY:
             menu_index = FILE;
             menu_tnormal(menu_addr, menu_index, 0);
             text_display();
             menu_bar(menu_addr, FALSE);
             break;

        case DUMP_KEY:
             menu_index = FILE;
             menu_tnormal(menu_addr, menu_index, 0);
             dump_display();
             menu_bar(menu_addr, FALSE);
             break;

        default:
             break;
```

Listing 10-4 (continued)

```
        }
        menu_tnormal(menu_addr, menu_index, 1);
        menu_bar(menu_addr, TRUE);
    }                               /* end keyboard handler */
}                                   /* end infinite loop */
    return;
}                                   /* end function */

/*************************************************
    Main Program
*************************************************/

main()
{
int  ap_id;                         /* application init verify */

WORD gr_wchar, gr_hchar,            /* values for VDI handle */
     gr_wbox, gr_hbox;

/*************************************************
    Initialize GEM Access
*************************************************/

    ap_id = appl_init();            /* Initialize AES routines */
    if (ap_id < 0)                  /* no calls can be made to AES */
    {                               /* use GEMDOS */
        Cconws('***> Initialization Error. <***\n');
        Cccnws('Press any key to continue.\n');
        Crawcin();
        exit(-1);                   /* set exit value to show error */
    }

    screen_phandle =                /* Get handle for screen */
        graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
    screen_vhandle = open_vwork(screen_phandle);
    set_screen_attr();              /* Get screen attributes */

/*************************************************
    Application Specific Routines
*************************************************/

    if (!rsrc_load('LISTER.RSC'))
    {
        form_alert(1, '[0][Cannot file LISTER.RSC file|Exiting ...] [OK]');
        exit(1);
    }

    control();
```

Listing 10-4 (continued)

```
/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     rsrc_free();
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
/*************************************************/
```

With regard to the various overhead used by LISTER, note that the LISTER.H header file is included in the application-specific data section of the program. The constants QUIT_KEY, DUMP_KEY, and TEXT_KEY represent the keyboard key codes for the menu shortcuts. The key code values are in Appendix C. The constants LINE_FEED, ESCAPE, and CR represent the line feed, escape, and carriage return characters. These characters are used to format the display. A line feed-carriage return combination indicates the end of a line in a file.

The global variable **max_lines** is set to the number of display lines per "page." As shown in Listing 10-4, LISTER only outputs the display to the screen. However, you can modify the program to output to the printer or another disk file. For screen output, the number of lines that can be displayed before the program must pause is 23. For printer output, the number of lines printed before a new page is required is 60 (assuming an 11-inch page length at six lines per inch). For disk output, there is no limit on the number of lines. If these alternate output destinations are implemented, the maximum line value must be able to change. Therefore, **max_lines** is a global variable rather than a preset constant.

The last bit of application-specific data consists of three character arrays called **def_search**, **sel_file**, and **file_name**. These arrays are used in conjunction with the **fsel_input( )** function. The **def_search** string contains the default search path to use when **fsel_input( )** is initiated. The **sel_file** string contains the default file name to be selected. When **fsel_input( )** returns to the calling program, **sel_file** contains the name of the selected file. To open a file, the complete path- and filename must be used. However, the pathname is contained in **def_search** and the filename is held in **sel_file**. These two strings are combined into a single filename in string **file_name**.

Function **main( )** is the same as in the last three programs. The resource file loaded for this program is LISTER.RSC. In the **control( )** function, **evnt_multi( )** waits for a keyboard or message event. Each type of event has its own section for processing. In the message event

section, only a menu selection is processed. Otherwise the message is ignored. A QUIT selection causes an exit from the infinite loop. Selection of the TEXTFILE display calls function **text_display( )** and then turns the menu bar off. The menu bar is turned off because the display function uses the entire screen, including the area for the menu bar. When the display is completed, the entire screen is cleared and the menu bar must be redisplayed. To ensure that the menu bar is displayed properly, it is turned off and then turned back on. The DUMPFILE selection calls function **dump_display( )** and turns the menu bar off. Case INFO is a request for more information, so **do_dialog( )** is called with the index to INFOBOX.

The keyboard event handler works similarly to the message event handler. Note that the keyboard event handler must explicitly change the menu title to its inverse state before processing.

Functions **text_display( )** and **dump_display( )** follow the same basic program flow. They get a selected filename from the user through function **get_file( )**. Next the selected file is opened in function **open_file( )**. Once the file is open, the display process can begin. Functions **text_display( )** and **dump_display( )** then hide the mouse, clear the screen, and perform their display functions in the appropriate format. When the end of the file is encountered, the file is closed, the screen is cleared again, and the mouse is turned back on. The two routines both pause the output display when the screen has been filled. The function **end_page( )** handles this condition.

The **get_file( )** routine uses the file selector manager. A call to **fsel_input( )** provides the user with a dialog box that allows the user to select a disk drive, a directory, and a particular file. The first parameter in **fsel_input( )** is the default search path. The second parameter is the default file to be selected. The default file and default directory are initialized in variables **def_search** and **sel_file** already mentioned above. Upon returning, **fsel_input( )** sets all three of its parameters. The parameter **exit_button** is set to a value indicating that the Exit button is selected. If **exit_button** is set to 0, the "Cancel" button is selected. Otherwise **exit_button** is set to 1 to indicate the OK button. The default search string and the default file selection string are set to the new values entered by the user.

After the file has been selected, a full filename specification needs to be created. Function **get_file( )** combines the returned search directory and selected filename so that a complete file and path description is set. The search path- and filename are combined into variable **file_name** so that a full file specification is available to the rest of the program.

Function **open_file( )** uses **fopen( )** to open the file. If an error occurs, **open_file( )** calls the AES function **form_error( )**. Function **form_error( )** displays an error box. The parameter to **form_error( )**

determines the error message displayed. This parameter must be a positive value. However, the variable **errno**, the global error variable used by the C compiler (see file ERRNO.H), contains the error number as a negative value. Therefore, the sign of the parameter to **form_error( )** is inverted.

Program LISTER has been written so that you can see how the file selector dialog box works and how an application is written to utilize events and the menu bar. As indicated earlier, you may want to add an option to have the output printed on the printer. You can implement this as a toggled menu selection using the CHECKED state to turn on and off the printer output. Other menu selections can also be added to set page length and page width of the printer output. Since program LISTER is relatively modular, you should be able to install these enhancements with little difficulty.

CHAPTER ELEVEN

Building a Better Mouse Trap

Dialog boxes, menus, and the file selector have all been handled by AES routines. This chapter provides a basic understanding of how the AES handles these user interactions. With the routines in the AES graphics library, you can emulate many of the interface features provided by GEM. For example, you can make slide bars, buttons, and dragging and sizing boxes. From this basis, you can then create your own interface features to customize your programs. To provide a graphics-based user interface, a program must have graphics routines and mouse access, both of which are provided by the graphics library.

Most of the routines in the graphics library control boxes in the GEM environment. A box is simply a rectangular image drawn on the screen. An outlined box is a lighter rectangular image drawn in half-intensity. The outline is used to indicate that some change can now occur on the selected box. For example, when you resize a window, an outlined image of the window is used to show the new window size. When you move a window, the outline of the window shows the new window location. When using the AES, a rectangular area is identified by the coordinates of its upper left corner, its width, and its height. All AES routines refer to rectangles in these terms. This is different from the VDI, which uses the coordinates of the upper left and lower right corners. This difference becomes important when you are programming with both the AES and the VDI.

There are ten graphics routines in the graphics manager: **graf_dragbox( )**, **graf_growbox( )**, **graf_handle( )**, **graf_mbox( )**, **graf_mkstate( )**, **graf_mouse( )**, **graf_rubberbox( )**, **graf_shrinkbox( )**, **graf_slidebox( )**, and **graf_watchbox( )**. Brief descriptions of each are provided below; for complete operational descriptions, see Appendix A.

The **graf_dragbox( )** function lets a user drag an outline within an application-defined boundary rectangle. For example, when you want to move an object in the resource editor program, you simply press the mouse button while on that object. A hand appears and you can drag the outline around. When you release the button, the object moves to the position of the outline. Function **graf_dragbox( )** controls the process of moving the outline under mouse control. When the user releases the mouse button, the function returns the x and y coordinates on the screen of the upper left corner of the box.

The **graf_growbox( )** function draws an expanding box outline. This is the function used by **form_dial( )** to show the expanding box. The function is given the size and position of the small box outline. This outline is drawn and then expands to the size and position of the large outline.

Function **graf_handle( )** has been used in all programs presented thus far and will continue to be used. This routine returns the handle to the current open screen workstation being used by GEM.

The **graf_mbox( )** function shows a box outline moving from one position to another. This function takes the initial size and location of the rectangle and the final location of the rectangle. The size of the box is not changed at all. The GEM documentation lists the name of this function as **graf_movebox( )**. The Megamax compiler has implemented this function with the name **graf_mbox( )**. Check your compiler manual to see which name is used for this function.

Function **graf_mkstate( )** returns the current mouse position, the state of the mouse buttons, and the state of the keyboard. Button and keyboard state values are returned in the same bit format as for the **evnt_multi( )** function described in Chapter 10.

The **graf_mouse( )** function lets the application change the mouse form to one of a predefined set of forms or to an application-defined form. This function also allows the application to hide and show the mouse, as done in program LISTER. The header file **gemdefs.h** defines the constant names for the different mouse forms (see Table 1-1).

The **graf_mouse( )** function has two parameters. The first parameter is one of the constant values identifying the mouse form to use. If the constant used is USER_DEF, the second parameter is used as a pointer to a mouse form definition block defined as structure MFORM in file **gemdefs.h** (see Figure 11-1). The MFORM structure has fields **mf_mask** and **mf_data** which are 16-by-16 bit arrays. The **mf_mask** array contains the mask bit map of the mouse image. The **mf_data** array defines the data bit map of the image. The mask and data bit maps are used to move the mouse around the screen in the same way the ball moves in program BOUNCE. The fields **mf_xhot** and **mf_yhot** determine the *hot spot* of the mouse. The hot spot is

Table 11-1: Mouse Form Constant Names

Constant Name	*Value*	*Form Image*
ARROW	0	Arrow
TEXT_CRSR	1	Text cursor (vertical bar)
HOURGLASS	2	Bumble bee
POINT_HAND	3	Hand with pointing finger
FLAT_HAND	4	Open hand
THIN_CROSS	5	Thin crosshair
THICK_CROSS	6	Thick crosshair
OUTLN_CROSS	7	Outlined crosshair
USER_DEF	255	User-defined image
M_OFF	256	Turn off mouse cursor
M_ON	257	Turn on mouse cursor

the point on the mouse image used to locate the mouse on the screen. For example, the hot spot for the arrow image would be at the tip of the arrow; the hot spot for the cross hair would be at the center of the cross; and the hot spot for a pointing hand would be at the tip of the finger. The x and y coordinates of the hot spot are measured in pixels relative to the upper left corner of the image. The **mf_nplanes** field indicates the number of planes in the bit map. The **mf_fg** and **mf_bg** fields specify the color index for the foreground and background colors respectively.

```
typedef struct mfstr {
    WORD        mf_xhot;
    WORD        mf_yhot;
    WORD        mf_yhot;
    WORD        mf_nplanes;
    WORD        mf_fg;
    WORD        mf_bg;
    WORD        mf_mask[16];
    WORD        mf_date[16];
} MFORM;
```

Figure 11-1 The MFORM Structure

The **graf_rubberbox( )** function draws a rectangle that can expand and contract with mouse movement. This is the routine that controls the outline when a window is resized. The upper left corner of the outline rectangle remains fixed, while the lower corner moves to determine the new size of the window.

The **graf_shrinkbox( )** performs the opposite function of **graf_growbox( )**. It draws a shrinking outline.

Function **graf_slidebox( )** keeps a sliding box object within its parent box. This is the routine used for the slide bars. The **graf_slidebox( )** function requires that the two boxes be objects and that the parent box contain the sliding box. Function **graf_slidebox( )** makes sure that the sliding box stays within the boundaries of the parent box.

The last graphics library routine, **graf_watchbox( )**, watches a rectangle while the user presses a button on the mouse. For example, move the mouse to the close box of a window. Press the left mouse button, and hold it down. As long as the button remains depressed and the mouse stays within the close box, the close box will be shown as selected. Now while holding the mouse button down, move the mouse outside the close box. As soon as the mouse exits the close box, the close box returns to its normal state. Function **graf_watchbox( )** is used to perform this action. When the mouse button is released, **graf_watchbox( )** returns a value indicating whether the mouse was inside or outside the box watched.

Program MOUSE

Program MOUSE demonstrates the graphics library functions, mouse events, and the use of a free-type tree in the resource file. The graphics library routines can be divided into five different categories. The first category deals with box movement such as growing, shrinking, moving, and sizing. The next category handles the sliding boxes. The third category is for the **graf_watchbox( )** routine. The fourth category includes mouse-handling functions. Finally, the fifth category is for the **graf_handle( )** routine. Each of the first four categories is demonstrated in a different area on the screen in program MOUSE. When the program is running, the user sees a box divided into four boxes (see Figure 11-2). The upper left area demonstrates moving and sizing boxes. The upper right area is for sliding boxes. The lower left area has a box that is watched. The lower right area shows the results of a mouse event.

The Resource File for Program MOUSE

The easiest way to create a screen layout is through the resource file. Therefore, the first thing to do is to create a resource file for program MOUSE. This resource file contains a menu bar and a free tree. The menu bar, named MAINMENU, is the default menu bar. In other words, it includes the Desk and File menus only. Under the Desk menu, the first entry should be changed to read "MOUSE Tester" and

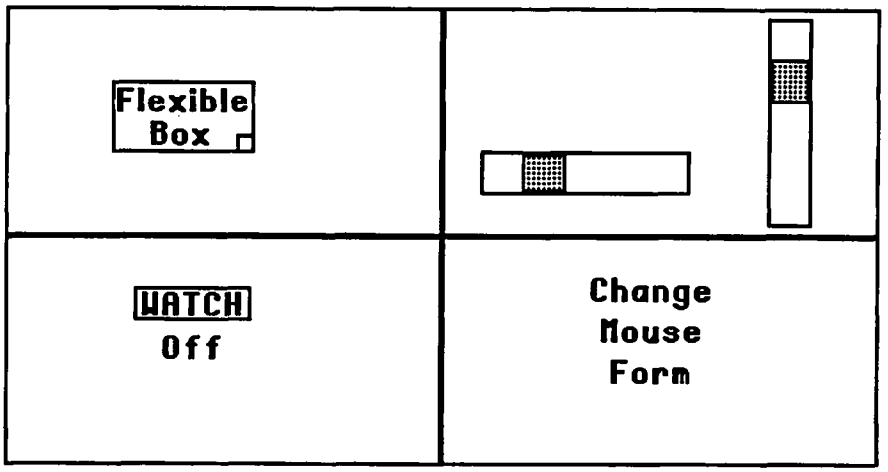


Figure 11-2 Program MOUSE Screen Layout

named INFO. The File menu has only the Quit selection, which should be named QUIT. Although there is no information dialog box for this program, the INFO menu selection must be named because if it is selected by the user, the program must be able to handle the selection (even if nothing is done).

The next tree to create for program MOUSE is a free tree. A free tree is used here because a dialog box causes all objects to be character-aligned. This restriction is not desired for MOUSE. To create a free tree, drag the free tree icon to the work area of the main window and name it MAINTREE. Open this free tree. You should have a box that is the root object in MAINTREE. Resize this box so that it fits within one-quarter of the screen. MAINTREE consists of a large box (the root) containing four smaller boxes. To create the four smaller boxes, drag a box from the parts box to the free tree box. Size and position this box to correspond with the upper left box shown in Figure 11-2; the exact size and position of the first box is not important. It would be rather cumbersome to try to make three more boxes exactly the same size as the first. As a shortcut, simply copy the first box. Place the mouse cursor over the box you just created. Hold down the Shift key, and press the mouse button. Now drag the box to the right so that it is next to the first box. When the copy is in position, release the mouse button. If you didn't position the new box exactly right, you can move it again. The Shift-Click combination causes the resource editor program to make a copy of the object. Both the RCS and MMRCP programs allow the user to copy objects in this manner. Repeat this procedure for the two lower boxes. When you have the four boxes, name the upper left box BOX0, the upper right box BOX1, the lower left box BOX2, and the lower right box BOX3.

BOX0 contains a box named FLEXBOX. Inside FLEXBOX are two STRING objects and a smaller box. The first string says "Flexible" and is named FLEXBOX1. The other string "Box" is named FLEXBOX2. The small box is named SIZEBOX. As you might guess, the objects in BOX0 are going to be used to demonstrate moving and sizing a rectangle.

In BOX1 are two slide bars. The horizontal large box is named HSLIDEIN, and the slider contained within is named HSLIDE. Note that the slider itself is filled with a pattern. The vertical slide bar is named VSLIDEIN; its slider is named VSLIDE. Make sure that the sliders fit inside the slide bars. The slider object must be a child of its slide bar. The objects in BOX1 are going to be used to demonstrate the **graf_slidebox( )** function.

BOX2 contains a button with the text "WATCH." This button is named WATCHBOX. Below it is a STRING object with text that says "Off." This string is named WATCHTXT.

BOX3 defines the area used for a mouse event. Inside BOX3 are three STRING objects for the text "Change," "Mouse," and "Form." These strings do not have any names.

The menu bar and the free tree alone make up this resource file. Figure 11-3 shows the object tree layout for the free tree. Once you have named all the objects, save this resource in file MOUSE.RSC.

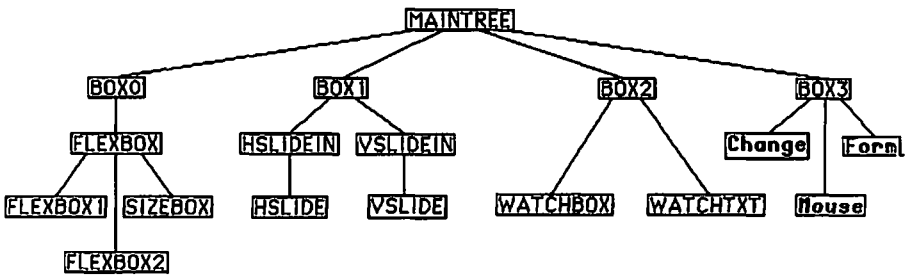


Figure 11-3 MOUSE.RSC Free Tree Layout

The Listing for Program MOUSE

In the application-specific data section of program MOUSE, the first statement includes the MOUSE.H file. Variable **tree_addr** holds the address of the free tree. The structure GRECT is the AES format for a rectangle. It contains field **g_x, g_y, g_w**, and **g_h** for the x and y coordinates and the width and height of a rectangle. The **box** array in program MOUSE holds the dimensions of boxes BOX0, BOX1, BOX2, and BOX3. This is why the exact position and size of these objects is not important in the resource file. Program MOUSE looks

in the OBJECT structure of these boxes and finds their sizes and positions on the screen.

In the GEM-related function section, the new routine **load_resource( )** has been added. This function is simply a generalized routine to load a resource file. If you want to provide for a more robust error-handling scheme, you can change **load_resource( )** instead of fiddling with function **main( )**. This separation of tasks holds to the modular design philosophy.

In function **main( )**, the program initialization and resource load is basically the same as in previous programs. The first application function is **initialize( )**, which initializes the global variables, the box array, and the tree address. Once the initialization is completed, the program moves to function **control( )**.

Function **initialize( )** first calls **hide_mouse( )** to hide the mouse cursor. Next **tree_addr** is set to the address of MAINTREE and the screen is cleared. The position of MAINTREE is changed to coordinates (10,20). When the resource editor creates a tree, the root is located at (0,0). If the root is left at this position, the image would conflict with the menu bar. Therefore, the coordinates of the root are moved down past the menu bar. Since the coordinates of all children of the root are placed with respect to the root, simply moving the root coordinates moves all of its children accordingly. This is a rather convenient feature.

Listing 11-1 Program MOUSE

```
/*********************************************************************
     MOUSE.C Mouse demonstration program

     This program shows how to use the mouse, event manager,
     object library, and graphics library.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */
#include  <errno.h>                /* errno declaration */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/
```

Listing 11-1 (continued)

```
/* Declare global arrays for VDI. */
typedef   int WORD;                 /* WORD is 16 bits */
WORD      contrl[12],               /* VDI control array */
          intout[128], intin[128],  /* VDI input arrays */
          ptsin[128], ptsout[128];  /* VDI output arrays */

WORD      screen_vhandle,           /* virtual screen workstation */
          screen_phandle,           /* physical screen workstation */
          screen_rez,               /* screen resolution 0,1, or 2 */
          color_screen,             /* flag if color monitor */
          x_max,                    /* max x screen coord */
          y_max,                    /* max y screen coord */
          m_hidden = FALSE;         /* mouse visibility status */

/*************************************************
     Application Specific Data
*************************************************/

#include  "mouse.h"

OBJECT    *tree_addr;               /* address of screen tree */
GRECT     box[4];                   /* 4 rectangles on screen */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;               /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)       /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;               /* use raster coords */
     new_handle = phys_handle;      /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
```

Listing 11-1 (continued)

```
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();        /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);   /* mono 2, color 0 or 1 */
}

hide_mouse()
/*************************************************
Function: Make mouse invisible if currently visible.
Input:    None. Uses variable m_hidden.
Output:   Sets m_hidden to TRUE.
*************************************************/
{
     if (!m_hidden)
     {
          graf_mouse(M_OFF, 0x0L);
          m_hidden = TRUE;
     }
}

show_mouse()
/*************************************************
Function: Make mouse visible if currently invisible.
Input:    None. Uses m_hidden.
Output:   None. Sets m_hidden to FALSE.
*************************************************/
{
     if (m_hidden)
     {
          graf_mouse(M_ON, 0x0L);
          m_hidden = FALSE;
     }
}

load_resource(rfile)
char *rfile;
/*************************************************
Function: Load resource file.
Input:    rfile     = string with resource file name.
Output:   Returns TRUE if file loaded, else FALSE.
*************************************************/
{
char temp[128];
```

Listing 11-1 (continued)

```
     if (!rsrc_load(rfile))
     {
          sprintf(temp, "[0][Cannot load file %s |Exiting ...] [OK]", rfile);
          form_alert(1, temp);
          return(FALSE);
     }
     return(TRUE);
}


/***********************************************
     Application Functions
***********************************************/


initialize()
/***********************************************
Function: Draw program screen.
Input:    Resource file must be loaded.
Output:   None.
***********************************************/
{
     hide_mouse();
     rsrc_gaddr(0, MAINTREE, &tree_addr);
     v_clrwk(screen_vhandle);
     tree_addr[MAINTREE].ob_y = 20;/* move down past menu bar */
     tree_addr[MAINTREE].ob_x = 10;
     objc_draw(tree_addr, 0, 10, 0, 0, x_max, y_max);

/* load rectangle coords */
     box[0].g_w = tree_addr[BOX0].ob_width;
     box[0].g_h tree_addr[BOX0].ob_height;
     objc_offset(tree_addr, BOX0, &box[0].g_x, &box[0].g_y);

     box[1].g_w = tree_addr[BOX1].ob_width;
     box[1].g_h = tree_addr[BOX1].ob_height;
     objc_offsst(tree_addr, BOX1, &box[1].g_x, &box[1].g_y);

     box[2].g_w = tree_addr[BOX2].ob_width;
     box[2].g_h = tree_addr[BOX2].ob_height;
     objc_offset(tree_addr, BOX2, &box[2].g_x, &box[2].g_y);

     box[3].g_w = tree_addr[BOX3].ob_width;
     box[3].g_h = tree_addr[BOX3].ob_height;
     objc_offset(tree_addr, BOX3, &box[3].g_x, &box[3].g_y);

     show_mouse();
     return;
}
```

Listing 11-1 (continued)

```
move_box()
/*************************************************
Function: Move a box.
Input:    None. Uses MAINTREE and global variables.
Output:   None. Moves box.
*************************************************/
{
WORD sx, sy,                        /* starting coords */
     fx, fy;                        /* final coords of box */

     graf_mouse(FLAT_HAND, 0x0L);  /* change mouse form */
     objc_offset(tree_addr, FLEXBOX, &sx, &sy);
     graf_dragbox(tree_addr[FLEXBOX].ob_width,
          tree_addr[FLEXBOX].ob_height,
          sx, sy,
          box[0].g_x, box[0].g_y, box[0].g_w, box[0].g_h,
          &fx, &fy);
     graf_mouse(ARROW, 0x0L);      /* restore form */

/* redraw parent box only to erase current position */
     objc_draw(tree_addr, BOX0, 0, 0, 0, x_max, y_max);
     graf_mbox(tree_addr[FLEXBOX].ob_width,
          tree_addr[FLEXBOX].ob_height,
          sx, sy, fx, fy);

/* set new location relative to parent */
     tree_addr[FLEXBOX].ob_x += (fx - sx);
     tree_addr[FLEXBOX].ob_y += (fy - sy);

/* redraw object */
     objc_draw(tree_addr, FLEXBOX, 10, 0, 0, x_max, y_max);
     return;
}

size_box()
/*************************************************
Function: Change size of box.
Input:    None. Uses MAINTREE and global variables.
Output:   None. Moves box.
*************************************************/
{
WORD sx, sy,                        /* starting x,y */
     new_width, new_height;         /* ending size */
static WORD    min_w = 0,           /* smallest size of box */
               min_h = 0;

/* set minimum values on first time through */
/* minimum values are initial values of FLEXBOX */
     if (min_w == 0 && min_h == 0)
     {
```

Listing 11-1 (continued)

```
            min_w = tree_addr[FLEXBOX].ob_width;
            min_h = tree_addr[FLEXBOX].ob_height;
        }

        graf_mouse(POINT_HAND, 0x0L); /* change mouse form */
        objc_offset(tree_addr, FLEXBOX, &sx, &sy);
        graf_rubberbox(sx, sy, min_w, min_h, &new_width, &new_height);
        graf_mouse(ARROW, 0x0L);      /* restore form */

/* redraw parent box only */
        objc_draw(tree_addr, BOX0, 0, 0, 0, x_max, y_max);

/* test if new size is greater than enclosing box */
        if ((new_width + tree_addr[FLEXBOX].ob_x) >
                tree_addr[BOX0].ob_width)
        {
            new_width =  /* if so, set to box edge */
                tree_addr[BOX0].ob_width - tree_addr[FLEXBOX].ob_x;
        }
        if ((new_height + tree_addr[FLEXBOX].ob_y) >
                tree_addr[BOX0].ob_height)
        {
            new_height =  /* if so, set to box edge */
                tree_addr[BOX0].ob_height - tree_addr[FLEXBOX].ob_y;
        }

/* set new width */
        tree_addr[FLEXBOX].ob_width = new_width;
        tree_addr[FLEXBOX].ob_height = new_height;

/* move size box */
        tree_addr[SIZEBOX].ob_x =
            tree_addr[FLEXBOX].ob_width - tree_addr[SIZEBOX].ob_width;
        tree_addr[SIZEBOX].ob_y =
            tree_addr[FLEXBOX].ob_height - tree_addr[SIZEBOX].ob_height;

/* redraw object */
        objc_draw(tree_addr, FLEXBOX, 10, 0, 0, x_max, y_max);
        return;
}


h_slide()
/*************************************************
Function: Change horizontal slider.
Input:    None. Uses MAINTREE and global variables.
Output:   None. Moves box.
*************************************************/
{
register long val;                  /* long needed for large widths */
```

Listing 11-1 (continued)

```
     val = graf_slidebox(tree_addr, HSLIDEIN, HSLIDE, 0);
/* find new x based on ratio: x = width * val / 1000 */
     val *= (tree_addr[HSLIDEIN].ob_width -
          tree_addr[HSLIDE].ob_width);
     tree_addr[HSLIDE].ob_x = val / 1000;
     objc_draw(tree_addr, HSLIDEIN, 10, 0, 0, x_max, y_max);
     return;
}

v_slide()
/*************************************************
Function: Change vertical slider.
Input:    None. Uses MAINTREE and global variables.
Output:   None. Moves box.
*************************************************/
{
register long val;                /* long needed for large heights */

     val = graf_slidebox(tree_addr, VSLIDEIN, VSLIDE, 1);
/* find new y based on ratio: y = height * val / 1000 */
     val *= (tree_addr[VSLIDEIN].ob_height -
          tree_addr[VSLIDE].ob_height);
     tree_addr[VSLIDE].ob_y = (val / 1000);
     objc_draw(tree_addr, VSLIDEIN, 10, 0, 0, x_max, y_max);
     return;
}

watch_box()
/*************************************************
Function: Change watch box.
Input:    None. Uses MAINTREE and global variables.
Output:   None. Moves box.
*************************************************/
{
/* if mouse released in box then change text */
     if (graf_watchbox(tree_addr, WATCHBOX, SELECTED, NORMAL))
     {
          if (strcmp(tree_addr[WATCHTXT].ob_spec, "Off") == 0)
               strcpy(tree_addr[WATCHTXT].ob_spec, "On ");
          else
               strcpy(tree_addr[WATCHTXT].ob_spec, "Off");
     }

     tree_addr[WATCHBOX].ob_state = NORMAL;
     objc_draw(tree_addr, BOX2, 10, 0, 0, x_max, y_max);
     return;
}

control()
/*************************************************
Function: Master control function.
```

Listing 11-1 (continued)

```
Input:    None. Program initialization must be done before
          entering this function.
Output:   None. Returns for normal program termination.
*************************************************/
{
OBJECT    *menu_addr;                /* address for menu */
WORD      mevx, mevy,                /* evnt_multi parameters */
          mevbut,
          keystate,
          keycode,
          mbreturn,
          msg_buf[8];                /* message buffer */
WORD      event,                     /* evnt_multi result */
          event_flag,                /* events to look for */
          m1_flag = 0,               /* 0 for entry to rectangle,
                                        1 for exit to rectangle */
          m1_x, m1_y, m1_w, m1_h,    /* rectangle for M1 event */
          menu_index;                /* keyboard menu title selected */

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);

/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* wait for a message indicating a menu selection */
     event_flag = (MU_BUTTON | MU_M1 | MU_MESAG);
for(;;)                             /* continue loop until quit */
{
     event = evnt_multi( event_flag,
          0,                         /* # mouse clicks */
          01,                        /* mouse buttons of interest */
          1,                         /* button state */
          m1_flag,                   /* first rectangle flags */
          box[3].g_x, box[3].g_y,    /* x,y of 1st rectangle */
          box[3].g_w, box[3].g_h,    /* height, width of 1st rect */
          0,                         /* second rectangle flags */
          0, 0,                      /* x,y of 2nd rect */
          0, 0,                      /* w,h of 2nd rect */
          msg_buf,                   /* message buffer */
          0, 0,                      /* low, high words for timer */
          &mevx, &mevy,              /* x,y of mouse event */
          &mevbut,                   /* button state at event */
          &keystate,                 /* status of keyboard at event */
          &keycode,                  /* keyboard code for key pressed */
          &mbreturn);                /* # times mouse key enter state */

     if (event & MU_MESAG)
     {
          if (msg_buf[0] != MN_SELECTED)      /* not a menu message */
               continue;                      /* then ignore */
```

Listing 11-1 (continued)

```
        if (msg_buf[4] == QUIT)
            break;              /* exit loop */

        switch(msg_buf[4])      /* find object index */
        {
        case INFO:              /* info selected */
            break;              /* ignore */
        default:
            break;
        }
                                /* reset title state */
        menu_tnormal(menu_addr, msg_buf[3], 1);
        menu_bar(menu_addr, TRUE);
    }                           /* end message handler */

    if (event & MU_BUTTON)
    {
        if (mevbut & 01)        /* button pressed down  */
        {                       /* find where event occurred */
            switch (objc_find(tree_addr, MAINTREE, 10, mevx, mevy))
            {
            case FLEXBOX:       /* the box itself */
            case FLEXBOX1:      /* 1st text line in box */
            case FLEXBOX2:      /* 2nd text line in box */
                move_box();
                break;

            case SIZEBOX:
                size_box();
                break;

            case HSLIDE:
                h_slide();
                break;

            case VSLIDE:
                v_slide();
                break;

            case WATCHBOX:
                watch_box();
                break;

            default:
                break;
            }
        }
    }                           /* end button handler */

    if (event & MU_M1)
    {
```

Listing 11-1 (continued)

```
        if (!m1_flag)           /* entered rectangle */
        {                       /* set to alternate mouse */
            graf_mouse(HOURGLASS, 0x0L);
            m1_flag = 1;        /* wait to exit */
            event_flag = MU_M1; /* only event */
        }
        else
        {
            graf_mouse(ARROW, 0x0L);
            m1_flag = 0;        /* wait for entry */
            event_flag = (MU_BUTTON | MU_M1 | MU_MESAG);
        }
    }
}                               /* end infinite loop */
    return;
}                               /* end function */

/***********************************************
    Main Program
***********************************************/

main()
{
int  ap_id;                     /* application init verify */

WORD gr_wchar, gr_hchar,        /* values for VDI handle */
     gr_wbox, gr_hbox;

/***********************************************
    Initialize GEM Access
***********************************************/

    ap_id = appl_init();        /* Initialize AES routines */
    if (ap_id < 0)              /* no calls can be made to AES */
    {                           /* use GEMDOS */
        Cconws("***> Initialization Error. (***\n");
        Cconws("Press any key to continue.\n");
        Crawcin();
        exit(-1);               /* set exit value to show error */
    }
    screen_phandle =            /* Get handle for screen */
        graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
    screen_vhandle = open_vwork(screen_phandle);
    set_screen_attr();          /* Get screen attributes */

/***********************************************
    Application Specific Routines
***********************************************/

    if (!load_resource("MOUSE.RSC"))   /* no resource file loaded */
        exit(1);
```

Listing 11-1 (continued)

```
     initialize();
     control();

/*************************************************
     Program Clean-up and Exit
*************************************************/

/* Wait for keyboard before exiting program */
     rsrc_free();
     v_clsvwk(screen_vhandle);      /* close workstation */
     appl_exit();                   /* end program */
}
/*************************************************/

/***************/
```

Once the new root coordinates are set, the screen can be drawn. The **objc_draw( )** function draws an object tree as seen earlier. In **initialize( ), objc_draw( )** starts at the root of MAINTREE and goes ten levels deep. The clipping rectangle used is the entire screen from (0,0) to (**xmax,ymax**). After the tree coordinates are set, the box coordinates can be determined. To use with the event library routines, these values must be in screen coordinates. However, the objects' coordinates are relative to their parents. The x and y coordinates of the object need to be converted to screen coordinates. Function **objc_offset( )** of the object library provides this feature. Given the tree address and the object index, **objc_offset( )** returns the x and y screen coordinates. The width and height remain the same because they are not affected by the position of the object. The coordinates of each of the four boxes is loaded into the array box. This completes the initialization and the mouse is turned back on.

Function **control( )** starts by locating the menu and displaying the menu bar. The parameters of **evnt_multi( )** warrant some closer examination. The **event_flag** variable holds the events that the program wants. When the mouse button is pressed, the user must be trying to select something on the screen. Therefore, a button event is required. When the mouse enters BOX3, the mouse form should change so a mouse event is also required. Finally, the user may select a menu item, which means a message event must also be included. The reason for using a variable as the event flag holder is that during normal processing, the application may want to look for different sets of events. By using a variable, the events accepted can be set while the program is running. This contributes to a more dynamic and modular design. For example, when the mouse enters BOX3, it can do nothing until it exits the box. Therefore, when the mouse leaves the box, the only event that must be waited on is a mouse event.

After the event flag has been set, the next three parameters for **evnt_multi( )** control the button event. The number of mouse clicks Is 0 because a single press is all that is required. Since the mouse button of interest is the left button only, the value 1 (bit 1 is set) is used for this parameter. The button state parameter indicates that the left button down is the state of interest (bit 1 is again set).

The next five parameters control the first mouse event. The variable **m1_flag** is initialized to 0 to indicate that a mouse event will occur when the mouse enters the area. The rectangle position and size are in element 3 of the box array. Since there is no second box of interest, the next five parameters are set to 0. You have already seen and used the remaining parameters in the previous programs.

Each event type is handled separately. If the event is a message, only menu messages are processed. All other types of messages are ignored. If the menu selection corresponds to QUIT, the infinite loop is exited. If the message corresponds to INFO, the message is ignored and the menu title is reset. If an information dialog box does exist, it can be displayed here. These are the only two menu selections that the program must look for.

A button event indicates that the user wants to manipulate something on the screen. When a button event occurs, the program checks which button is pressed. Actually, only button 1 should cause an event based upon the parameters for **evnt_multi( )**. After the button number is checked, the program needs to find where the mouse was located when the button was pressed. It obviously was not pressed in the menu bar, because the AES would already have handled that. It could have been pressed somewhere in the work area. To find if the mouse was over an object, function **objc_find( )** is used. Function **objc_find( )** searches through a tree to see if any objects are located at the button down event position. This function can start searching from any object in the tree (not just the root) and can continue for any number of levels. If **objc_find( )** finds an object at the given location, it returns the index of the object. If no objects in the tree are located under the coordinates given, **objc_find( )** returns −1. In MOUSE, **objc_find( )** is used as the control variable for a switch statement. The various cases are selected based upon the object index.

If the user wants to move the box in BOX0, the mouse must be located on object FLEXBOX. However, FLEXBOX1 and FLEXBOX2 cover part of FLEXBOX so these two objects also cause a move condition to be activated. To move the box, application function **move_box( )** is called. If the user wants to resize this box in BOX0, the mouse must be located inside SIZEBOX. To change the size of the box, application function **size_box( )** is called.

To move the horizontal slider, the user places the mouse over the HSLIDE object and presses the mouse button. In this case, application function **h_slide( )** is called. If the mouse is located on the vertical

slider, **v_slide( )** is called. If the mouse is in object WATCHBOX, **watch_box( )** is called. Any other objects or −1 is processed through the default case and ignored.

The final event accepted by **event_multi( )** is a mouse event. This occurs when the mouse enters BOX3. If **m1_flag** is 0, the event occurred when the mouse entered the rectangle. Therefore, the program changes the mouse form to an alternate mouse form (here, to the HOURGLASS, which is actually the bumble bee on the Atari ST). The **graf_mouse( )** function changes the image. The program now waits until the mouse exits this box; all other events are ignored. Therefore the **event_flag** is set to MU_M1 only and **m1_flag** is set to 1 (to wait for an exit). When the mouse exits the box, another mouse event occurs. In this case, **m1_flag** is set to 1. The program changes the mouse back to the arrow form, the **m1_flag** is set to 0, and the **event_flag** has the button, mouse, and message events.

Now that all the events have been handled, look at the application functions. Start with function **move_box( )**. When a box is going to be moved, the GEM interface requires the program to change the mouse form to indicate the function being performed. While moving a box, the mouse form should be the open hand. The first statement in **move_box( )** changes the mouse form to a FLAT_HAND. While the user is positioning the box, your program should display an outline. The **graf_dragbox( )** function is used to drag an outline of the box around on the screen. Function **graf_dragbox( )** needs to know the starting coordinates of the box in *screen* coordinates. Thus, **objc_offset( )** is used to get the starting screen coordinates (the box's current position). Function **graf_dragbox( )** is then called. Its parameters are the width and height of the box, the starting coordinates, a boundary rectangle, and two variables to hold the final coordinates. The boundary rectangle limits the area in which the outline may travel. In MOUSE, the box should not move outside of BOX0. When the user releases the mouse button, **graf_dragbox( )** returns with its last two parameters containing the final coordinates of the box. At this point, the mouse form must be restored to the arrow form and the program needs to update the screen.

There are two methods for updating the screen. The first method redraws just the area affected. First, the program redraws BOX0. This causes all the contents of BOX0 to be erased (actually written over). Next, it draws a box moving from the old position to the new position. Then it redraws the moved box at its new location. Program MOUSE uses this first method. To redraw just BOX0, function **objc_draw( )** is used. The drawing starts at BOX0 and continues for 0 levels, which causes just BOX0 to be drawn. Then function **graf_mbox( )** is used to show a moving outline. Next the new object coordinates are set. Since there is no function to go from screen to relative coordinates, the

easiest way to handle the problem is to change the object's coordinate by the distance moved. Finally, **objc_draw( )** is called again to draw the FLEXBOX and all its children at the new location.

The other method would be to calculate the object's new relative x and y position. Use **graf_mbox( )** to show a moving box, and then call **objc_draw( )** to draw BOX0 and all of its children. Because BOX0 is drawn first (parents are drawn before their children), it erases the area. Then FLEXBOX is drawn at its new location. It is a matter of convenience and preference to go one way or another.

When changing the size of an object, the user places the mouse at the lower right corner of the object and presses the mouse button. The mouse form changes to a pointing hand, a rubber-box outline appears, and the user can change the size of the object. A program may place restrictions on the minimum and maximum size of the box. In MOUSE, the minimum size of the box is the height of the two text lines and the width of the two lines. The maximum size is set so that the box does not exceed the limits of BOX0. Once the new size has been set and verified, a growing box outline is drawn and the object is redrawn with its new size.

In function **size_box( )**, the first statements determine the minimum size of the box. The program could use the width and height values of the text strings to calculate the minimum size. However, MOUSE assumes that the original size of FLEXBOX is the minimum size to use. You can go back and change the resource file to set FLEXBOX to any minimum size you desire. To set the minimum size, **size_box( )** uses the static variables **min_w** and **min_h**. On the first time through **size_box( )**, these variables are set at 0 and FLEXBOX is still at its original size. If this is the case, **min_w** and **min_h** are set to the original width and height of FLEXBOX. Once the minimum values are set, the mouse form is changed to the pointing hand. Function **objc_offset( )** is used to get the screen coordinates of FLEXBOX. Then **graf_rubberbox( )** is called to allow the user to set the new size. Function **graf_rubberbox** uses the minimum width and height values but does not have any parameters for a maximum width and height. The maximum size must be calculated manually, which is done a few statements later. When the user releases the mouse button, **graf_rubberbox( )** returns and function **size_box( )** resets the mouse form to the arrow.

The same general redrawing method is used here that is used in **move_box( )**. First, BOX0 only is redrawn to erase the area. Then the new size is tested to see if it exceeds the maximum boundaries. If the new size does exceed the limits, the value is truncated at the edge of the boundary. For example, if the right edge of FLEXBOX extends past the right edge of BOX0, the right edge of FLEXBOX is set to stop at the right edge of BOX0. Note how the relative object coordinates are

used for this calculation. The relative x coordinate of FLEXBOX plus its new width must be less than the width of BOX0. A similar test is done with the relative y and new height.

Once the new width and height are set, the relative coordinates of SIZEBOX (the small box in FLEXBOX) must be changed. Otherwise, it will appear in the same *relative* position as before. The size box should always be in the lower right corner. When a box is resized, the relative position of the lower right corner changes. Therefore, the new relative x and y coordinates of SIZEBOX are calculated. Now that all the children of FLEXBOX are set, **objc_draw( )** is called once again to draw FLEXBOX in its new size.

For the slide bars, the user places the mouse on the slider and presses the mouse button. The slider can then be moved to any position within the slide bar. Control of the slider is handled by function **graf_slidebox( )**. The tree address, the index of the slide bar object, and the index of the slider object are passed to **graf_slidebox( )**. The last parameter determines whether the box moves in the vertical or horizontal direction. 0 indicates horizontal movement and 1 indicates vertical control. The **graf_slidebox( )** function automatically keeps the object inside its parent. When the user releases the mouse button, the function returns a value between 0 and 1000. This indicates the relative position of the *center* of the slider inside the slide bar. A 0 is the furthest left and 1000 is the furthest right position on a horizontal slide bar and the top and bottom of a vertical slide bar, respectively. On a horizontal slide bar, once the new relative position is known, the new x position must be calculated. Since the x position is relative to the parent, a ratio can be set up as this:

$$\frac{\text{new x}}{\text{slide bar width}} = \frac{\text{value returned}}{1000}$$

Solving this equation for the new x coordinate gives this:

new x = (value returned * slide bar width) / 1000

For a vertical slide bar, the new y coordinate is calculated with the following equation:

new y = (value returned * slide bar height) / 1000

After calculating the new x or y coordinate of the slider, the value must be set in the OBJECT structure for the slider. Then the slide bar is redrawn.

Function **h_slide( )** handles the request for change to the horizontal slider. Function **graf_slidebox( )** is used to control the slider move-

ment. When **graf_slidebox( )** returns, variable **val** has the relative position of the slider in the slide bar. Variable **val** is declared as a long integer because the multiplication may exceed the range of a standard integer. Note that the width of the slide bar is adjusted by subtracting the width of the slider. The value returned by **graf_slidebox( )** is the relative position of the *center* of the slider. The center of the slider never reaches the edge of the slide bar. At best, the center of the slider gets to within one-half slider width of the end of the slide bar. One-half width at the left edge and one-half width at the right edge means that there is one slider's width of area on the slide bar that the center of the slider cannot reach. Therefore, the width of the slider must be taken out because this width is a "dead zone" in which the slider cannot move. After calculating the new x coordinate, the slide bar (and its child, the slider) can be redrawn using **objc_draw( )**. Function **v_slide( )** is much the same except that it uses the y coordinate and the object's height. It also adjusts the total height of the slide bar by subtracting the height of the slider itself.

The last application function is **watch_box( )**. This function uses the **graf_watchbox( )** function to see if an object button is selected. The parameters of **graf_watchbox( )** provide the address of the tree, the index of the object being watched, and the state of the object when the mouse is in and out of the box. The **graf_watchbox( )** function returns TRUE when the mouse button is released while the mouse is inside the box and FALSE otherwise. The result of releasing the button inside the box here is that the text is toggled from "On" to "Off" or vice versa. When **graf_watchbox( )** returns, it leaves the watched box in its last state. Thus, if you are emulating a button as done in MOUSE, then when **graf_watchbox( )** returns the box needs to be set back to its original state.

This completes program MOUSE. There are quite a few changes you can make to this program. First, you might try adding an information dialog box as used in the previous four programs. Also, function **size_box( )** did not use the **graf_growbox( )** function to show the change of size for the box. Use the **move_box( )** function as an example of where to insert the **graf_growbox( )** function. Another area to investigate is the mouse form. After you have worked with the predefined mouse forms, create one of your own using the MFORM structure. Look back at the raster programs for examples of creating bit maps and masks.

If your program changes the mouse form, it is responsible for restoring the mouse form to the arrow when the mouse leaves your work area. For example, a word processing program uses the TEXT_CRSR mouse form. If the user moves the mouse to one of the window control areas, the desktop, or the menu bar, the word processor is responsible for changing the mouse form back to the arrow.

The easiest way to handle this in a program is to set up a mouse event. When the mouse leaves the work area, the mouse event occurs and your program can change the mouse form. While the mouse is outside the work area, your program waits for the mouse to enter the work area (the reverse procedure as shown in program MOUSE for BOX3). When the mouse returns to the work area, the mouse form can be set to whatever form is required. This should be done to remain consistent with the GEM interface.

In this chapter, you have seen some more events, particularly mouse and button events, and some of the basic routines used by the AES in its graphics interface. These routines are useful when you want to provide graphic tools for the user to set such as the control panel accessory. The next chapter shows the use of windows, which is the primary use of many of these graphic routines.

CHAPTER TWELVE

Windows on the World

By now, you have probably noticed that the programs written so far lack one primary aspect of the GEM environment: the use of windows. Most programs have used just the desktop window, which is the default location for most output functions accessed by the programs.

The use of windows in a program provides a more natural working environment. A window allows the user to separate tasks and organize them on the screen in any order. For example, a word processor may allow two or three different windows to be open with each window editing a different file. When the user is working on one file and needs information from another file, it is then a simple matter to take the information needed from one window and transfer it to the original window.

There are a number of different situations that the program must deal with when using a window. This chapter covers the use of windows in a program, how a program can create and provide a window for the user, and how the program must communicate with the AES to handle window operations. Finally, two demonstration programs show the use of the many window concepts that are discussed.

Window Rules

Chapter 9 introduced the components of a window. These components are shown again in Figure 12-1. The components include the close box, full box, title bar and move bar, scroll bars, sliders, arrows,

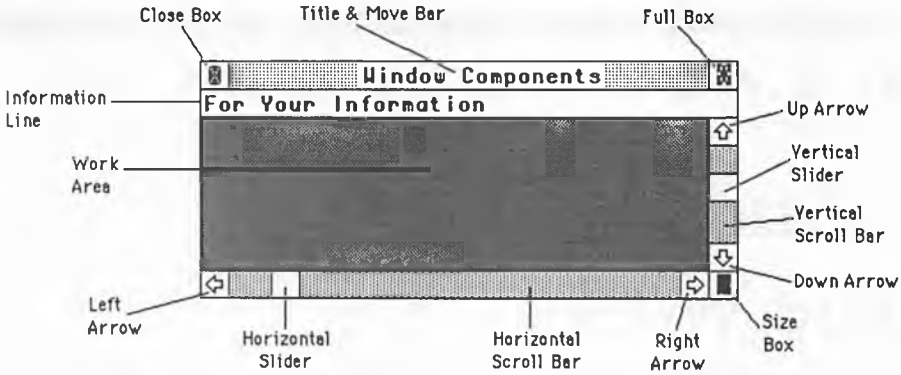


Figure 12-1 Window Components

size box, and work area. The title bar and work area are the only required components for a GEM window. All other components may or may not appear depending on the application that creates the window. To quickly review the components, the user selects the close box to indicate to the application that the window is to be closed and removed from the screen. The full box toggles the size of the window between the largest possible size and the normal size of the window. The size box allows the user to change the dimensions of the window. The move bar allows the user to move the window on the desktop. The scroll bar, sliders, and arrows move the window over the data being displayed. The arrows move the window up and down or left and right. The size of the slider relative to the scroll bar is proportional to the size of the window relative to the total size of the file.

When an application is running, it is responsible for the contents of the work area. All user interactions with the remaining components of the window are handled by the AES. As mentioned earlier, when the user interacts with one of these components, called *window control areas*, the AES sends a message to the application indicating that the user has changed the window environment in some way.

The Window Manager

Using a window in a program requires the use of the functions contained in the window library of the AES. This library contains the routines that create windows, manipulate them, and set their attributes. Accessing a window is similar to using a workstation. Each window has a *window handle* that identifies the window to the AES. GEM can handle up to eight different windows at one time.

The desktop itself is a window even though it doesn't look or work

like a normal window. The menu bar can be thought of as the information line and the gray desktop as the work area. The desktop window always has a handle with the value 0 (window handles are always integers). The desktop uses one window so there are seven windows available for applications and desk accessories. Most applications use no more than four windows at one time. Otherwise, if the application were to use all seven remaining windows, there would be no windows for the desk accessories to use. Even the desktop limits the number of windows that can be open at one time. For example, by double-clicking the floppy disk icon you can open a window to show the disk's directory. You can keep opening windows for the same floppy disk. However, if you try to open another window once you have opened four windows, the desktop gives you a message saying that no more windows are available.

Window Procedures

When a window is opened, the AES must reserve space in memory for this window to keep track of the window's attributes. Therefore, when you want to use a window, you must follow a particular sequence of function calls to initiate and use the window. These calls create the window, open the window, close it, and finally delete it when it is no longer needed. Creating the window tells the AES to reserve memory space for a new window. The creation process also defines the size and components of the window. Opening the window causes the AES to draw the window on the screen. The AES takes care of drawing the control areas. The work area is left blank to be filled in by the application. When the window is no longer needed on the screen, the program must close the window. This removes the window from the screen. Any data underneath the window is then displayed. If the window is no longer needed, it is deleted from the AES so that another window is available for a program or desk accessory.

In GEM, the window that is completely visible is called the topmost, or active, window. This is the window currently being accessed by the user. Other windows are partially or completely hidden by the active window. These other windows remain inactive until the topmost window is closed or the user selects one of the inactive windows. If the topmost window is closed or removed from the screen, the window just below it becomes the active window. If the user clicks another window, that window becomes the topmost window.

Because of the number of different conditions that might occur, programs with windows must be event-driven. Once the program has created and opened a window, it waits for messages from the AES.

When the user interacts with a window control area, the AES sends a message informing the program of the request. The program then updates the window according to the user's requests. The program is responsible for any user interaction with the work area.

Window Manager Routines

The window manager (window library) consists of a set of routines that allow an application to manipulate a window. These routines include **wind_create( )**, **wind_open( )**, **wind_close( )**, **wind_delete( )**, **wind_calc( )**, **wind_get( )**, **wind_set( )**, **wind_find( )**, and **wind_update( )**.

Function **wind_create( )** allocates space for a window and returns a handle to that window. The **wind_open( )** function opens a window on the desktop to a specified size. In addition, this function generates an event that causes the application to draw the work area. The **wind_close( )** function simply closes a window and removes it from the screen. This function also causes the appropriate events to occur so that the images underneath the window (which are now visible) can be redrawn. Function **wind_delete( )** removes a window from the AES and allows the handle to be reused by another application.

There are two rectangles associated with a window that are of interest to an application. The first rectangle is the work area. The work area rectangle is simply the part of the window labeled as the work area (see Figure 12-1). As the window is moved and sized, the work area rectangle changes its size and position. The second rectangle, the border area is the total area of the window including the work area and all the control areas. Function **wind_calc( )** calculates the border area given the work area of the window and vice versa.

Functions **wind_get( )** and **wind_set( )** allow an application to get and change information about a specific window. Function **wind_get( )** returns such information as the position and size of the work area or the border area, the size of a full window (that is, when the full box is selected), the position of the sliders, the size of the sliders, and the handle to the active window. Function **wind_set( )** is the opposite of **wind_get( )**. It allows the application to change the window's appearance such as changing its title, changing the contents of the information line, setting the position and size of the entire window, and setting the sliders and slider sizes.

Function **wind_find( )** returns the handle of a window under the coordinates given as its parameters. Function **wind_update( )** tells the AES that the application is about to update a window or has completed the update. This function is used so that when the program is updating a window, the program does not receive any other events that would change the window's appearance. This stops all window

processing until the application has completed redrawing the window. **Wind_update( )** also allows an application to take control of the mouse functions or return control to the AES. Taking control of the mouse functions means that the application program must handle the processes normally performed by the screen manager such as following the mouse, controlling the menus, and controlling the window components.

The sample programs presented in this chapter utilize most of these functions. The first program, WINDOW1, is a general introduction to using windows in a program. The second program, WINDOW2, shows how an application handles the remaining components. Each program is designed to allow you to experiment with the window functions.

Window Messages

As mentioned above, a program manages its windows on an event basis; that is, an event determines the next process that needs to be performed. The AES communicates to the application through message events. These messages are generated when the application's window becomes the new top window, the application's window has been closed, the full box is selected, and so forth. Essentially, a message is sent in response to any user interaction with a window control area or when the work area needs to be redrawn. The contents of the message simply inform the application of what the user has requested. For example, for the application to make a new top window, the application must set that window as the top window using **wind_set( )**. The AES then responds by redrawing that window's control areas as the top window and issuing a redraw message to the application owning the window (that is, the application that created the window).

A close message requires the application to close the window. Opening or closing a window may also require other aspects to be changed. For example, when the last window is closed in an editor program, the Save or Save As.. options should be disabled because there is no more data to save.

The full box selection requires the application to determine whether the window is at its full size or not. If it is at its full size, the application must use **wind_set( )** to reset the size of the window to its original smaller size. If the window is at its smaller size, the application must reset the window to its full size. If sliders are used on the window, changing the size of the window also requires that the relative size of the sliders be changed. As the window's size

increases or decreases, the size of the scroll bar also increases or decreases. The slider size must be changed to maintain the correct proportions.

An arrow message indicates that the user has selected one of the directional arrows at the ends of the scroll bars. When an arrow message is received, the application must move the window one unit in the direction of the arrow. For a text screen, this would be one line up or down or one character left or right. For a spreadsheet application, this would be one row up or down or one column left or right. If the scroll bar is clicked (this is also indicated by an arrow message), the window should move one screen width at a time. If the sliders are moved, the application needs to calculate what portion of the data to display.

Moving or resizing a window such as the rubber-box and drag-box outlines is handled by the AES for the graphic display. When the user releases the mouse button, the move or resize message is sent to the application. The message contains the new position or size of the window. The application should check that the window has not become too small. The **wind_set( )** function is also used here to set the new position or size. If the size of the window is changed, the size of the sliders must be checked.

Finally, a redraw message indicates that a portion of the window work area has now become visible. The application is responsible for filling this newly visible area.

Redrawing a Window

A window on the screen may have a visible portion. Any such visible portion of a window can be represented by a set of nonintersecting rectangles. Figure 12-2 shows a set of five rectangles that represent the visible portion of window B. The AES maintains a list of nonintersecting rectangles that cover the visible portion of an open window. A redraw message will be issued for each window. A redraw message contains the area of the screen to be redrawn and the handle of the window to redraw. For example, if the top window in Figure 12-2 is closed, there would be a rectangular area (the area previously covered by the window) to be redrawn. Closing the window causes the AES to issue a redraw message to the application(s) owning windows A and B. The same application may own both windows A and B but not necessarily. It is the responsibility of the application program to then redraw the work area portion of the window.

When the top window is closed, the visible area for window B increases. Therefore, the AES changes the visible rectangle list (see

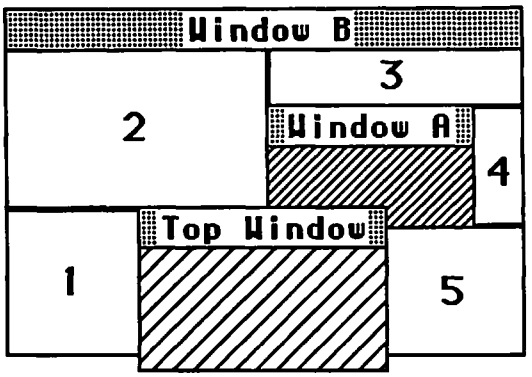


Figure 12-2 Window B's Visible Rectangle List

Figure 12-3) and sends a redraw message to the application owning this window. To draw a window, only those visible portions should be output to the screen. This can be accomplished by setting the clipping rectangle to one of the visible rectangles in the rectangle list. Anything outside of the clipping rectangle would not be drawn. By going through each rectangle in the list, the entire visible portion of a window can be drawn. However, it could become quite time-consuming for an application to redraw an entire window when only a small portion needs to be redrawn as in Figure 12-3. To speed the process of redrawing a window, the clipping rectangle is set to the intersection of the redraw area and the visible rectangle. In the figure, rectangle 1 intersects with the redraw area in the lower right corner. The clipping rectangle is set to this intersection. When the window is redrawn, only the image in this intersection is output. Rectangles 2, 3, and 4 do not intersect with the redraw area at all so no redrawing is done. The left half of rectangle 5 intersects with the redraw area

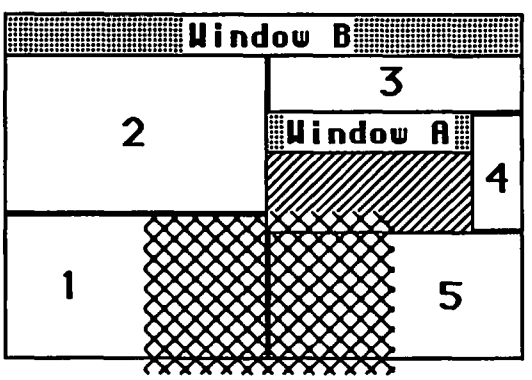


Figure 12-3 Window B's Visible Rectangle List and the Redraw Area

so this portion of the window is output. Therefore, the redraw procedure consists of intersecting each visible rectangle with the redraw area. If an intersection exists, the clipping rectangle is set to the intersection area and the window is drawn. If no intersection exists, the next visible rectangle is used. When all visible rectangles in the list have been checked, the window is redrawn.

An application usually has just one function used to redraw the window. This one function may consist of many other functions; yet, it provides a single entry point to the window-drawing process. By having a single drawing function, the redraw sequence can make just one call when the window needs to be redrawn. Otherwise, the redraw process becomes cluttered with too many function calls and the flow of the program becomes obscured. As you can see, dealing with windows is not the simplest of tasks and having a clear program flow aids you tremendously when changing or debugging.

The WINDOW Structure

The AES maintains an internal data structure to handle window displays. Through the **wind_get( )** function, you can retrieve various values such as the position and size of the work or border area, the position and size of the full size window, the position and size of the horizontal and vertical sliders, the handle of the top window, the first rectangle in a window's visible rectangle list, and the next rectangle in the window's visible rectangle list. However, neither **wind_get( )** nor any other window functions provides any further access to window values needed for the processing of the program. Specifically, once the window is created, you cannot retrieve information about which control areas are visible, whether the window is visible or not (that is, opened or closed), and whether the window is at its full size or not. All these values must be maintained by the application.

To help keep track of these window values, the programs in this book use a structure called WINDOW. (See Figure 12-4.) The WINDOW structure is provided by the application. It is *not* a part of GEM. The WINDOW structure holds information about a window not readily accessible through an AES function. In addition to the WINDOW structure, there is an array of WINDOW structures called windows. This array holds up to the maximum number of windows specified by the application. For the programs in this book, the maximum number of windows is four. When a window is created by the application, the window's information must be stored in the array. When a window is deleted by the application, it must also be deleted from the windows array. Handling the **windows** array requires some additional

```
typedef struct wind_type {
    WORD        handle;
    WORD        mf_type;
    WORD        mf_visible;
    WORD        mf_fullsize;
} WINDOW;
```

Figure 12-4 The WINDOW Structure

overhead. However, once you see the convenience of having this array, you will agree that it is necessary.

The handle field contains the handle of the window. The type field contains the components available for this window. Each bit in the field represents a different component. The bit layout is discussed later. The visible field keeps track of whether the window is open or closed. If visible is TRUE, the window is open. The field fullsize is TRUE if the window is at its full size.

Program WINDOW1

You should now be ready to create a window-exercising application. Program WINDOW1 is the first step toward creating an application outline file much like OUTLINE.C for the VDI. Of course, before creating a complete application, you need a resource file.

The WINDOW1 Resource File

The resource file for program WINDOW1 is shown in Figure 12-5. It contains two trees. There is a menu named MAINMENU and a dialog box called INFOBOX. The dialog box, as shown in the figure, is much like the dialog boxes shown earlier. The menu contains the Desk and File titles and a Windows title. Under the Desk menu, the first entry should be changed to read "About WINDOW1." Name this selection as INFO. Under the File menu, there is the option Quit to be named QUIT. The Windows menu has four entries, each to open one of four windows. The entries are named WIND1, WIND2, WIND3, and WIND4. Once you have created this resource file, save it under the name WINDOW1.RSC.

Overview of WINDOW1

Program WINDOW1 demonstrates how an application handles windows. This program allows the user to open up to four windows at

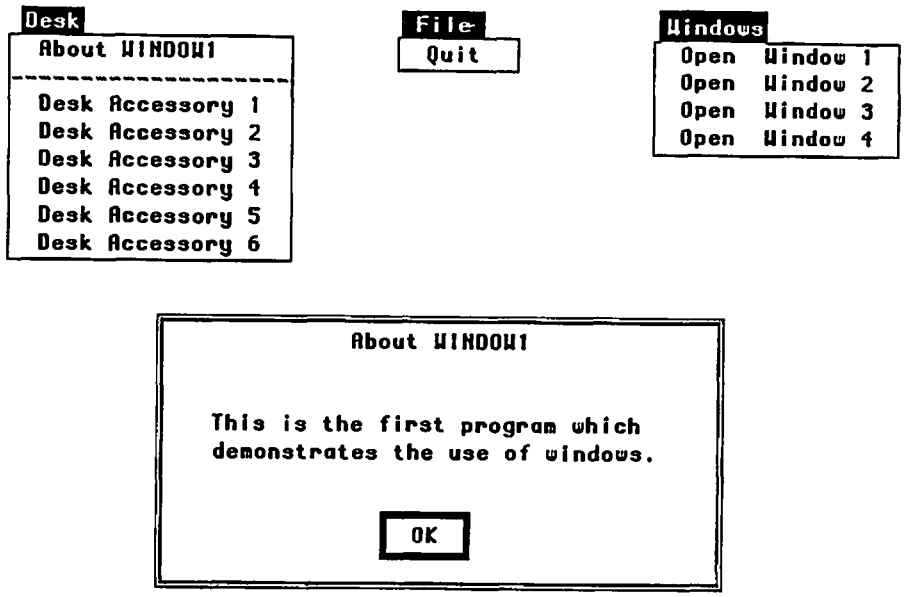


Figure 12-5 Program WINDOW1 Resource File

any one time. The contents of the windows are filled ovals with each window using a different fill pattern. The user may move, resize, close, reopen, and full size any of the windows. When the user quits the program, any open windows are closed and all windows are deleted.

The WINDOW1 program is shown in Listing 12-1. Do not be alarmed by the length of this program. You have already seen most of the functions and procedures in previous programs. Next is a description of some of the organization changes made to this program.

Listing 12-1 Program WINDOW1

```
/*********************************************************************
     WINDOW1.C Sample window application

     This program demonstrates the use of windows within
     an application.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>              /* Standard IO */
#include  <osbind.h>             /* GEMDOS routines */
#include  <gemdefs.h>            /* GEM AES */
```

Listing 12-1 (continued)

```
#include  <obdefs.h>                /* GEM constants */
#include  <errno.h>                 /* errno declaration */

#define   FALSE     0
#define   TRUE      !FALSE

/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;                /* WORD is 16 bits */
WORD      contrl[12],               /* VDI control array */
          intout[128], intin[128],  /* VDI input arrays */
          ptsin[128], ptsout[128];  /* VDI output arrays */

WORD      screen_vhandle,           /* virtual screen workstation */
          screen_phandle,           /* physical screen workstation */
          screen_rez,               /* screen resolution 0,1, or 2 */
          color_screen,             /* flag if color monitor */
          x_max,                    /* max x screen coord */
          y_max,                    /* max y screen coord */
          m_hidden = FALSE;         /* mouse visibility status */

WORD      gr_wchar, gr_hchar,       /* values for system sizes */
          gr_wbox, gr_hbox;
                                    /* set min window size */
#define   WMIN_WIDTH     (2*gr_wbox)
#define   WMIN_HEIGHT    (3*gr_hbox)

WORD      ap_id;                    /* application ID */

WORD      num_windows;              /* number of windows open */
WORD      cur_window;               /* index for current window */

typedef struct wind_type {          /* window record */
     WORD      handle;              /* AES window handle */
     WORD      type;                /* window attribute bits */
     WORD      visible;             /* flag if visible */
     WORD      fullsize;            /* flag if full size */
} WINDOW;

#define   MAX_WINDOW 4              /* allow for 4 windows */
WINDOW    windows[MAX_WINDOW];      /* array of window records */
WINDOW    desktop;                  /* desktop record */
WORD      xdssk, ydesk,             /* desktop area */
          wdesk, hdesk;

/*************************************************
     Application Specific Data
*************************************************/
```

Listing 12-1 (continued)

```
#include  'window1.h'              /* resource header file */

WORD      wind1, wind2,            /* window indices */
          wind3, wind4;

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
          work_out[57],
          new_handle;                    /* handle of workstation */
int       i;

     for (i = 0; i < 10; i++)            /* set for default values */
          work_in[i] = 1;
     work_in[10] = 2;                    /* use raster coords */
     new_handle = phys_handle;           /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];

     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}

hide_mouse()
/*************************************************
Function: Make mouse invisible if currently visible.
Input:    None. Uses variable m_hidden.
```

Listing 12-1 (continued)

```
Output:   Sets m_hidden to TRUE.
*************************************************/
{
     if (!m_hidden)
     {
          graf_mouse(M_OFF, 0x0L);
          m_hidden = TRUE;
     }
}

show_mouse()
/*************************************************
Function: Make mouse visible if currently invisible.
Input:    None. Uses m_hidden.
Output:   None. Sets m_hidden to FALSE.
*************************************************/
{
     if (m_hidden)
     {
          graf_mouse(M_ON, 0x0L);
          m_hidden = FALSE;
     }
}

load_resource(rfile)
char *rfile;
/*************************************************
Function: Load resource file.
Input:    rfile    = string with resource filename.
Output:   Returns TRUE if file loaded, else FALSE.
*************************************************/
{
char temp[128];

     if (!rsrc_load(rfile))        /* error loading file */
     {                             /* set alert format */
          sprintf(temp, "[0][Cannot load file %s |Exiting ...] [OK]", rfile);
          form_alert(1, temp);     /* show alert box */
          return(FALSE);
     }
     return(TRUE);
}

do_dialog(box_index)
WORD box_index;
/*************************************************
Function: Display a dialog box centered on the screen.
Input:    box_addr  = index of dialog box
Output:   Returns index of object used for exit.
*************************************************/
{
```

Listing 12-1 (continued)

```
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of box */
     rsrc_gaddr(0, box_index, &box_addr);

/* get size and location of a box centered on screen */
     form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
     smallx = xbox + (wbox / 2);
     smally = ybox + (hbox / 2);
     smallw = 0;
     smallh = 0;

/* reserve area on screen for box display */
     form_dial(FMD_START,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw an expanding box */
     form_dial(FMD_GROW,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* draw dialog box */
     objc_draw(box_addr, 0, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
     exit_object = form_do(box_addr, 0);

/* draw a shrinking box */
     form_dial(FMD_SHRINK,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
     form_dial(FMD_FINISH,
          smallx, smally, smallw, smallh,
          xbox, ybox, wbox, hbox);

/* reset exit object state to unselected */
     box_addr[exit_object].ob_state = NORMAL;

     return(exit_object);
}

set_clip(x, y, w, h)
WORD x, y, w, h;
```

Listing 12-1 (continued)

```
/*************************************************
Function: Set the clipping rectangle.
Input:    x    = x coord of clip rect
          y    = y coord of clip rect
          w    = width (in pixels) of clip rect
          h    = height (in pixels) of clip rect
Output:   None. Sets new clipping rectangle.
*************************************************/
{
WORD clip[4];

     clip[0] = x;
     clip[1] = y;
     clip[2] = x + w - 1;
     clip[3] = y + h - 1;
     vs_clip(screen_vhandle, TRUE, clip);
}

init_windows()
/*************************************************
Function: Initialize global window values.
Input:    None.
Output:   None. Sets global window values.
*************************************************/
{
int  i;

     num_windows = 0;              /* no windows open */
     for (i = 0; i < MAX_WINDOW; i++)
          windows[i].handle = -1;  /* set records to unused */
     desktop.handle = 0;           /* desktop is always 0 */

                                   /* get desktop work area */
     wind_get(desktop.handle, WF_WORKXYWH,
          &xdesk, &ydesk, &wdesk, &hdesk);
     return;
}

create_window(newkind, wtitle)
WORD      newkind;
char      *wtitle;
/*************************************************
Function: Open a new window.
Input:    newkind   = window attributes to include
          wtitle    = string for window title
Output:   Returns index if window created, else -1
Notes:    Info line, and slider values are NOT set
          by this function. Full window size is set to
          desktop work area.
*************************************************/
{
WORD new;
```

Listing 12-1 (continued)

```
/* check if windows are available */
     if (num_windows >= MAX_WINDOW)      /* max windows in use */
     {
          form_alert(1, "[0][Maximum number of windows reached.][OK]");
          return(-1);
     }

/* find window record to use */
     for(new = 0; new < MAX_WINDOW; new++)
          if (window[new].handle < 0)     /* record found */
               break;
     if (new >= MAX_WINDOW)               /* no records available */
     {
          form_alert(1, "[0][No window records found available.][OK]");
          return(-1);
     }

/* create window for AES */
     windows[new].handle =
          wind_create(newkind, xdesk, ydesk, wdssk, hdesk);

     if (windows[new].handle < 0)        /* AES could not make window */
     {
          form_alert(1, "[0][AES error opening a window.|Cannot continue.][OK]");
          return(-1);
     }

/* fill window record */
     windows[new].type = newkind;
     windows[new].fullsize = FALSE;
     windows[new].visible = FALSE;

/* set window title */
     wind_set(windows[new].handle, WF_NAME, wtitle, 0, 0);

     nwm_windows++;                  /* add window to count */
     return(new);                    /* return index */
}

open_window(wi_index)
/*************************************************
Function: Make a window visible.
Input:    wi_index  = index to window
Output:   None. Sets work area and visibility flag in window record.
*************************************************/
{
WORD xsize, ysize, wsize, hsize;

/* check if already open */
     if (windows[wi_index].visible)
          return;
```

Listing 12-1 (continued)

```
/* get current window size */
     wind_get(windows[wi_index].handle, WF_PREVXYWH,
          &xsize, &ysize, &wsize, &hsize);

/* if no size, then must be first time opened */
     if ( !(xsize || ysize || wsize || hsize) )
     {
          xsize = xdesk;          /* defawlt to desktop work area */
          ysize = ydesk;
          wsize = wdesk;
          hsize = hdesk;
     }

/* open window */
     hide_mouse();
     graf_growbox((xsize + wsize/2), (ysize + hsize/2),
          gr_wbox, gr_hbox, xsize, ysize, wsize, hsize);
     wind_open(windows[wi_index].handle, xsize, ysize, wsize, hsize);
     windows[wi_index].visible = TRUE;  /* flag window is open */
     show_mouse();
}

close_window(wi_index)
/*************************************************
Function: Close a window.
Input:    wi_index  = index to window to close
Output:   None. Closes window and set visibility flag.
*************************************************/
{
WORD xsize, ysize, wsize, hsize;

/* check if already closed */
     if (!windows[wi_index].visible)
          return;

/* get current window area */
     wind_get(windows[wi_index].handle, WF_CURRXYWH,
          &xsize, &ysize, &wsize, &hsize);

/* close window */
     hide_mouse();
     wind_close(windows[wi_index].handle);
     graf_shrinkbox((xsize + wsize/2), (ysize + hsize/2),
          gr_wbox, gr_hbox, xsize, ysize, wsize, hsize);
     windows[wi_index].visible = FALSE; /* flag window as closed */
     show_mouse();
}

del_window(wi_index)
WORD wi_index;
/*************************************************
Function: Delete window from AES
```

Listing 12-1 (continued)

```
Input:    wi_index  = index of window to delete
Output:   None. Window deleted from AES and windows.
*************************************************/
{
     if (windows[wi_index].visible)      /* still on screen */
          close_window(wi_index);        /* so close it */

/* delete from AES */
     wind_delete(windows[wi_index].handle);
/* set record as available */
     windows[wi_index].handle = -1;
/* remove window from count */
     num_windows--;
}

find_window(wi_handle)
WORD wi_handle;
/*************************************************
Function: Find index for window record with given handle.
Input:    wi_handle = handle of window to search for
Outpwt:   Returns index or -1 if not found.
*************************************************/
{
register  int i;

     for (i = 0; i < MAX_WINDOW; i++)
          if (windows[i].handle == wi_handle)
               return(i);
     return(-1);
}

do_redraw(wi_index, wi_redraw, x, y, w, h)
WORD wi_index,
     (*wi_redraw)(),
     x, y, w, h;
/*************************************************
Function: Redraw all clipping rectangles.
Input:    wi_index  = index to window being updated
          wi_redraw = address of function used to draw window
          x, y      = redraw area X,Y coord
          w, h      = redraw area width and height
Output:   None. Screen is updated.
Notes:    wi_redraw cannot use any parameters or return any values.
          You may create a do_redraw function for each window,
          or pass the parameters through global variables.
*************************************************/
{
GRECT     redraw,                  /* redraw area */
          clip;                    /* current clip area */

     hide_mouse();
     wind_update(TRUE);            /* freeze window status */
```

Listing 12-1 (continued)

```
     redraw.g_x = x;              /* set redraw area */
     redraw.g_y = y;
     redraw.g_w = w;
     redraw.g_h = h;

/* get first clip rectangle */
     wind_get(windows[wi_index].handle, WF_FIRSTXYWH,
          &clip.g_x, &clip.g_y, &clip.g_w, &clip.g_h);

/* begin redraw loop until no more clip rectangles */
     while (clip.g_w && clip.g_h)
     {
          if (rc_intersect(&redraw, &clip))
          {
               set_clip(clip.g_x, clip.g_y, clip.g_w, clip.g_h);
               (*wi_redraw)();     /* particular redraw function */
          }
          wind_get(windows[wi_index].handle, WF_NEXTXYWH,
               &clip.g_x, &clip.g_y, &clip.g_w, &clip.g_h);
     }
     wind_update(FALSE);          /* screen is ready */
     show_mouse();
}


/************************************************
     Application Functions
************************************************/


draw_function()
/************************************************
Function: Draw in current window.
Input:    None. cur_window is index to current window to draw in.
Output:   None. Draws in window.
************************************************/
{
WORD temp[4];
WORD xwork, ywork, wwork, hwork;

     wind_get(windows[cur_window].handle, WF_WORKXYWH,
          &xwork, &ywork, &wwork, &hwork);
     vsf_interior(screen_vhandle, 2);
     vsf_style(screen_vhandle, 8);
     vsf_color(screen_vhandle, 0);
     temp[0] = xwork;
     temp[1] = ywork;
     temp[2] = temp[0] + wwork - 1;
     temp[3] = temp[1] + hwork - 1;
     v_bar(screen_vhandle, temp);  /* clear work area */

     if (cur_window == wind1)
          vsf_style(screen_vhandle, 9);
```

Listing 12-1 (continued)

```
     else if (cur_window == wind2)
          vsf_style(screen_vhandle, 10);
     else if (cur_window == wind3)
          vsf_style(screen_vhandle, 11);
     else if (cur_window == wind4)
          vsf_style(screen_vhandle, 12);
     vsf_color(screen_vhandle, 1);
     v_ellipse(screen_vhandle, xwork + wwork/2, ywork + hwork/2,
          wwork/2, hwork/2);
     return;
}

control()
/*************************************************
Function: Master control function.
Input:    None. Program initialization must be done before
          entering this function.
Output:   None. Returns for normal program termination.
*************************************************/
{
OBJECT    *menu_addr;              /* address for menu */
WORD      end_program = FALSE;     /* exit flag */
WORD      mevx, mevy,              /* evnt_multi parameters */
          butstate,
          mevbut,
          keystate,
          keycode,
          mbreturn,
          msg_buf[8];              /* message buffer */
WORD      event,                   /* evnt_multi result */
          menu_index;              /* keyboard menu title selected */
WORD      draw_function();         /* screen drawing function */

/* initialize window indices */
     wind1 = -1;
     wind2 = -1;
     wind3 = -1;
     wind4 = -1;

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);

/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* initial button status to wait for */
     butstate = TRUE;             /* button down */

/* wait for a message indicating a menu selection */
do                                /* continue loop until quit */
{
```

Listing 12-1 (continued)

```
event = evnt_multi( (MU_KEYBD | MU_MESAG | MU_BUTTON),
     0,                        /* # mouse clicks */
     0,                        /* mouse buttons of interest */
     bwtstate,                 /* button state */
     0,                        /* first rectangle flags */
     0, 0,                     /* x,y of 1st rectangle */
     0, 0,                     /* height, widht of 1st rect */
     0,                        /* second rectand flags */
     0, 0,                     /* x,y of 2nd rect */
     0, 0,                     /* w,h of 2nd rect */
     msg_buf,                  /* message buffer */
     0, 0,                     /* low, high words for timer */
     &mevx, &mevy,             /* x,y of mouse event */
     &mevbut,                  /* button state at event */
     &keystate,                /* status of keyboard at event */
     &keycode,                 /* keyboard code for key pressed */
     &mbreturn);               /* # times mouse key enter state */

wind_update(TRUE);             /* hold window processing */

if (event & MU_MESAG)
{
 switch(msg_bwf[0])
 {
 case MN_SELECTED:             /* menu chosen */
     switch(msg_buf[4])
     {
     case QUIT:                /* exit program */
          end_program = TRUE; /* set exit flag */
          break;

     case INFO:                /* display program info */
          do_dialog(INFOBOX);
          break;

     case WIND1:               /* open/close window 1 */
          if (wind1 < 0)       /* window must be created */
          {
               wind1 = create_window(
                    (NAME|CLOSER|FULLER|MOVER|SIZER),
                    " Window 1 ");
               if (wind1 < 0) /* error creating window */
               {
                    end_program = TRUE;
                    break;
               }
          }
          if (windows[wind1].visible)
          {                    /* window is open so close it */
               close_window(wind1);
               menu_text(menu_addr, WIND1, "  Open  Window 1");
          }
```

Listing 12-1 (continued)

```
            else
            {                       /* window is closed so open it */
                 cpen_window(wind1);
                 menu_text(menu_addr, WIND1, '  Close Window 1');
            }
            break;

       case WIND2:                  /* open/close window 2 */
            if (wind2 < 0)          /* window must be created */
            {
                 wind2 = create_window(
                      (NAME|CLOSER|FULLER|MOVER|SIZER),
                      ' Window 2 ');
                 if (wind2 < 0) /* error creating window */
                 {
                      end_program = TRUE;
                      break;
                 }
            }
            if (windows[wind2].visible)
            {                       /* window is open so close it */
                 close_window(wind2);
                 menu_text(menu_addr, WIND2, '  Open  Window 2');
            }
            else
            {                       /* window is closed so open it */
                 open_window(wind2);
                 menu_text(menu_addr, WIND2, '  Close Window 2');
            }
            break;

       case WIND3:                  /* open/close window 3 */
            if (wind3 < 0)          /* window must be created */
            {
                 wind3 = create_window(
                      (NAME|CLOSER|FULLER|MOVER|SIZER),
                      ' Window 3 ');
                 if (wind3 < 0) /* error creating window */
                 {
                      end_program = TRUE;
                      break;
                 }
            }
            if (windows[wind3].visible)
            {                       /* window is open so close it */
                 close_window(wind3);
                 menu_text(menu_addr, WIND3, '  Open  Window 3');
            }
            else
            {                       /* window is closed so open it */
                 open_window(wind3);
                 menu_text(menu_addr, WIND3, '  Close Window 3');
            }
            break;
```

Listing 12-1 (continued)

```
        case WIND4:                 /* open/close window 1 */
             if (wind4 < 0)         /* window must be created */
             {
                  wind4 = create_window(
                       (NAME|CLOSER|FULLER|MOVER|SIZER),
                       " Window 4 ");
                  if (wind4 < 0) /* error creating window */
                  {
                       end_program = TRUE;
                       break;
                  }
             }
             if (windows[wind4].visible)
             {                      /* window is open so close it */
                  close_window(wind4);
                  menu_text(menu_addr, WIND4, "  Open  Window 4");
             }
             else
             {                      /* window is closed so open it */
                  open_window(wind4);
                  menu_text(menu_addr, WIND4, "  Close Window 4");
             }
             break;

        default:
             break;
        }
                                    /* reset menu title */
        menu_tnormal(menu_addr, msg_buf[3], 1);
        menu_bar(menu_addr, TRUE);
        break;                      /* end MN_SELECTED */

        case WM_REDRAW:             /* redraw windows */
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;                 /* no window listed */
        do_redraw(cwr_window, draw_function,
             msg_buf[4], msg_buf[5], msg_bwf[6], msg_buf[7]);
        break;

      case WM_NEWTOP:               /* new window is on top */
      case WM_TOPPED:
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;                 /* no window listed */
        wind_set(windows[cur_window].handle, WF_TOP, 0, 0, 0, 0);
        break;

      case WM_CLOSED:               /* close box pressed */
        if ( (cur_windcw = find_window(msg_buf[3])) < 0)
             break;                 /* no window listed */
        close_window(cur_window);
        if (cur_window == wind1) /* set appropriate menu item */
             menu_text(menu_addr, WIND1, "  Open  Window 1");
        else if (cur_window == wind2)
```

Listing 12-1 (continued)

```
                menu_text(menu_addr, WIND2, '  Open  Window 2');
            else if (cur_window == wind3)
                menu_text(menu_addr, WIND3, '  Open  Window 3');
            else if (cur_window == wind4)
                menu_text(menu_addr, WIND4, '  Open  Window 4');
            break;

        case WM_FULLED:                 /* full box pressed */
            if ( (cur_window = find_window(msg_buf[3])) < 0)
                break;                  /* no window listed */
            if (windows[cur_window].fullsize)
            {                           /* full to regular size */
            WORD newx, newy, neww, newh;
                                        /* get previous window size */
                wind_get(windows[cur_window].handle, WF_PREVXYWH,
                    &newx, &newy, &neww, &newh);
                wind_set(windows[cur_window].handle, WF_CURRXYWH,
                    newx, newy, neww, newh);
            }
            else
            {                           /* regular to full size */
            WORD xfull, yfull, wfull, hfull;
                                        /* get full window size */
                wind_get(windows[cur_window].handle, WF_FULLXYWH,
                    &xfull, &yfull, &wfull, &hfull);
                wind_set(windows[cur_window].handle, WF_CURRXYWH,
                    xfull, yfull, wfull, hfull);
            }
            windows[cur_window].fullsize ^= TRUE;
            break;

        case WM_ARROWED:                /* arrow and slide bars not used */
        case WM_HSLID:
        case WM_VSLID:
            break;

        case WM_SIZED:                  /* window resized or moved */
        case WM_MOVED:
            if ( (cur_window = find_window(msg_buf[3])) < 0)
                break;                  /* window not listed */
            if (msg_buf[6] < WMIN_WIDTH)
                msg_buf[6] = WMIN_WIDTH;
            if (msg_buf[7] < WMIN_HEIGHT)
                msg_buf[7] = WMIN_HEIGHT;
            wind_set(windows[cur_window].handle, WF_CURRXYWH,
                msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7]);
            break;

          default:
            break;
        }                               /* end message switch */
        }                               /* end message handler */
```

Listing 12-1 (continued)

```
     if (event & MU_KEYBD)
     {
      switch(keycode)
      {
/*     case QUIT_KEY:
          menu_index = FILE;
          menu_tnormal(menu_addr, menu_index, 0);
          end_program = TRUE;
          break;
*/
        default:
          break;
      }
      menu_tnormal(menu_addr, menu_index, 1);
      menu_bar(menu_addr, TRUE);
     }                              /* end keyboard handler */

     if (event & MU_BUTTON)         /* button handler */
     {
          butstate = !(butstate);
     }                              /* end button handler */

     if (end_program)               /* close open windows */
     {                              /* and delete all windows */
     int  i;

          for (i = 0; i < MAX_WINDOW; i++)
               if (windows[i].handle >= 0)
                    del_window(i);
     }

     wind_update(FALSE);            /* resume processing */

} while (!end_program);             /* program control loop */

return;
}                                   /* end function */

/*************************************************
     Main Program
*************************************************/

main()
{

/*************************************************
     Initialize GEM Access
*************************************************/

     ap_id = appl_init();           /* Initialize AES routines */
     if (ap_id < 0)                 /* no calls can be made to AES */
```

Listing 12-1 (continued)

```
    {                               /* use GEMDOS */
        Cconws("***> Initialization Error. <***\n");
        Cconws("Press any key to continue.\n");
        Crawcin();
        exit(-1);                   /* set exit value to show error */
    }

    screen_phandle =                /* Get handle for screen */
        graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
    screen_vhandle = open_vwork(screen_phandle);
    set_screen_attr();              /* Get screen attributes */

/*************************************************
    Application Specific Routines
*************************************************/

    if (!load_resource("WINDOW1.RSC"))/* no resource file loaded */
        exit(1);
    init_windows();
    graf_mouse(ARROW, 0x0L);
    control();

/*************************************************
    Program Clean-up and Exit
*************************************************/

    rsrc_free();
    v_clsvwk(screen_vhandle);       /* close workstation */
    appl_exit();                    /* end program */
}
/*************************************************/
/******/
```

The system header files have not changed. The GEM application overhead contains the usual VDI declarations. Note that the variables **gr_wchar, gr_hchar, gr_wbox,** and **gr_hbox** have been moved from function **main( )** to this section of the program. The variable **gr_wchar** has the value of the width of one character cell in pixels. The variable **gr_hchar** contains the height of one character cell in pixels. These values are used in the next two **#define** statements for the minimum window width, WMIN_WIDTH, and minimum window height, WMIN_HEIGHT. The minimum width will be at least two characters and the minimum height will be at least three characters. Variable **ap_id** has also been moved to the GEM application overhead because if the program needs to send a message, element 1 of the message buffer must contain the ID number of the application originating the message. The remainder of the GEM application overhead is used to facilitate the windows array mentioned above.

The use of the **windows** array in your application program is optional. It is used here because it is one of the most straightforward and simplest methods for maintaining multiple windows in an application. These declarations have been placed under the GEM application overhead section in case you decide to use WINDOW1 or WINDOW2 as an outline for your own applications.

The variable **num_windows** keeps track of how many windows have been opened by the application. Variable **cur_window** is the index into the array **windows** for the currently open window. Next comes the definition of the WINDOW structure. The constant MAX_WINDOW determines the maximum number of windows that may be *created* by this program at any one time. GEM provides for only eight windows. This program allows up to four windows to be created. The array **windows** is then declared as an array of WINDOW structures. Variable **desktop** is a WINDOW structure that keeps track of the desktop window. This is done so that accessing the desktop window remains consistent with accessing the other windows within the program. The four variables **xdesk, ydesk, wdesk,** and **hdesk** hold the work area of the desktop window. The desktop work area is the gray portion of the screen below the menu bar. In general, windows should not be placed outside of the desktop work area and their size should not exceed the size of this area.

Moving to the GEM-related functions, the first several functions have already been discussed. These are: **open_vwork( ), set_screen_attr( ), hide_mouse( ), show_mouse( ), load_resource( ),** and **do_dialog( ).** The next function, **set_clip( ),** provides an interface between the AES rectangle and the VDI rectangle. To set the clipping rectangle, the VDI **vs_clip( )** function must be used. However, most of the coordinates for rectangular areas are in the AES format consisting of the coordinates for the upper left corner, the width, and the height. These values need to be converted to the VDI rectangle format consisting of the coordinates for opposite corners. Function **set_clip( )** provides this interface and sets the clipping rectangle.

The function **init_windows( )** initializes the **windows** array and gets the values of the desktop window. Function **init_windows( )** should be called during the initialization process of the program as in the call to **set_screen_attr( )** in function **main( ).** The **init_windows( )** function sets the number of windows in use to 0 (since no windows have been created yet) and sets the handle for each window in the windows array to −1. Any window created by the AES has a positive handle. Setting the handle to −1 indicates that this element in the **windows** array is not being used. The desktop window handle is initialized to 0. To set the remaining desktop window values, **init_windows( )** makes a call to **wind_get( ).** The first parameter of **wind_get( )** is the handle of the window being queried. The second

parameter is a constant value that indicates the type of information being requested. The constant names and values are listed in Appendix A. The next four parameters are used to hold the returned values. In this call to **wind_get( )**, the program is requesting the size and location of the desktop work area.

The next four functions in this section of the program are **create_window( )**, **open_window( )**, **close_window( )**, and **del_window( )**. These functions maintain the **windows** array and provide the appropriate procedures for doing the described action. Much like accessing an object in a tree array, your program is accessing windows through the **windows** array. Function **create_window( )** creates a window in the AES if there is room. If not, an alert box is issued to the user and the function returns. The **create_window( )** function returns the index of the array element for the window created. If the window is not created, a −1 is returned. Just as an object is accessed through an index, a window is also accessed through an index.

The **create_window( )** function first checks if the maximum number of windows are in use. Then the function tries to locate an element in the windows array not currently in use. An element not in use is indicated by a handle element less than 0 (usually −1). Variable **new** is set to the index of the array that holds the new window. If **new** exceeds the value MAX_WINDOW, there is a problem in the **windows** array and an alert message is given. If **new** contains the proper index, a window can be created. Function **wind_create( )** is called and returns a handle to the newly created window. The first parameter to **wind_create( )** specifies the control areas to include in the window. The next four parameters define the largest size that the window may assume. Because a window's size may not exceed the desktop work area, the largest size assumed in this program is the entire desktop work area. If the calling program needs a smaller full-size window, the full-size window area can be set through the **wind_set( )** function. The values used for **wind_create( )** only limit the absolute maximum size that the window may achieve. Function **wind_create( )** returns the handle of the new window. If the handle returned is less than 0, the AES had a problem creating the window and an alert box indicates the situation.

The element in the **windows** array is then filled. The type field is set to the attributes of the window. The full-size field starts at FALSE. The visible field is also set to FALSE because creating a window does not open the window. Opening a window must be done explicitly. Next function **wind_set( )** is called to set the title bar of the window. Finally, variable **num_windows** is incremented by 1 so that the window count is accurate. The index to the newly created window is returned to the calling program.

Function **open_window( )** takes the index of a window and opens

that window. The process of opening a window simply requires a call to function **wind_open( )**. However, to make a program look more sophisticated, opening a window should consist of hiding the mouse (to avoid any screen drawing conflicts), drawing a growing box outline, opening the window, and showing the mouse. Function **open_window( )** can be called at any time after a window has been created but should not be called when the window is already open. Therefore, the first statement in **open_window( )** tests if the window is visible. If it is, the function just returns. If not, **open_window( )** continues the opening process.

Two conditions can occur when a window is opened. Either the window has never been opened before or it is being reopened. In the latter case, it would make sense that the window be opened to the same size it was when it was closed. A program can request the previous size of a window through the **wind_get( )** function shown in **open_window( )**. The constant WF_PREVXYWH indicates that the previous position and size of the window are desired. The information is returned into variables **xsize, ysize, wsize,** and **hsize**. If the window has never been opened before, these variables all have the value of 0. The next statement in **open_window( )** tests for this condition. If the window has never been opened before, its default starting size is the size of the desktop work area. If a previous size is available, these values are used. To actually open the window, function **hide_mouse( )** is called followed by the **graf_growbox( )** function. Next the window is opened through **wind_open( )** and the mouse is redisplayed with **show_mouse( )**.

Function **hide_mouse( )** is called before processing the screen to avoid conflicts with the screen manager. When the mouse is drawn, the image under the mouse is saved in memory. When the mouse is moved, the screen manager replaces this image. If **hide_mouse( )** is not called, when the mouse is moved, the screen manager replaces the contents of the screen under the mouse before the window was drawn. You have probably noticed this situation in some of the earlier programs where the screen was cleared to a white background, and when the mouse was moved a small rectangular patch of gray was drawn in its place. Turning off the mouse causes the screen manager to reset the saved image.

The next function, **close_window( )**, closes a window. If the window is already closed, the function returns. The first function call is to **wind_get( )** to get the current position and size of the window (using the constant WF_CURRXYWH). These values are used by the **graf_shrinkbox( )** function. To close the window, **hide_mouse( )** is called first, the shrinking outline box is drawn, the window is closed, the visible flag is set to FALSE, and **show_mouse( )** is called.

Deleting a window in **del_window( )** is even easier than closing a

window. The first statement here checks if the window is currently visible on the screen. If it is, the window needs to be closed before it can be deleted from the AES. If the window were deleted from the AES before being erased from the screen, the only way to remove the window's image would be to erase the entire screen and redraw all the open windows. To avoid this problem, if the window is still visible **del_window( )** calls **close_window( )** before continuing. Once the window is closed, it can be removed from the AES with the **wind_delete( )** function. Next the handle for the element in the **windows** array is set to −1 to indicate that this array element is now available again. Finally, the number of windows in use is decremented by 1.

There is one other function needed to utilize the **windows** array. The AES refers to a window by its handle. When using the windows array, a window is referenced by its index. Therefore, an application needs a function to find which element in the windows array has the specified handle. Function **find_windows( )** does this search. Given the handle for a window, it searches through the **windows** array and returns the index to the **windows** array for the element that has that handle. If the handle is not found, −1 is returned.

The last function discussed in this section is **do_redraw( )**. This function is called whenever the program receives a window redraw message. Essentially, **do_redraw( )** implements the redrawing process discussed earlier. The parameters of the **do_redraw( )** function are the index of the window to redraw, the address of the function that redraws the window, and the redraw area. No values are returned. Two rectangles, called **redraw** and **clip**, are used in function **do_redraw( )**. Rectangle **redraw** corresponds to the redraw area, passed in the parameters **x**, **y**, **w**, and **h**. Rectangle clip is set to the intersection of the visible rectangle and the redraw rectangle.

Function **do_redraw( )** starts with a call to **hide_mouse( )** for the same reason **hide_mouse( )** is called when opening and closing a window. Next, **wind_update( )** is called with a parameter of TRUE. This tells the AES that the application is updating its windows and that it should not process any user interactions with the screen. This prevents a backlog of events from occurring and ensures that the screen remains current. Next the redraw rectangle is initialized.

The redraw process loops through all visible rectangles in a windows rectangle list. The first rectangle on the visible rectangle list is retrieved using **wind_get( )** with constant WF_FIRSTXYWH in its parameter list. The first rectangle is then returned into the next four parameters. A while loop processes each visible rectangle. If the width and height for a visible rectangle are both 0, all rectangles have been processed and the loop can terminate.

Within the loop, the first call is to function **rc_intersect( )**. This function returns TRUE if the two rectangles in its parameter list

intersect with one another. Otherwise FALSE is returned. If the rectangles do intersect, the function calculates the intersection rectangle and returns its position and size in the second parameter. Thus, the second parameter is changed by **rc_intersect( )** to describe the intersection rectangle.

If the visible rectangle and the redraw area do intersect, the clipping rectangle is set to the intersection and the window is redrawn. The while loop continues by using **wind_get( )** to retrieve the next rectangle in the window's visible rectangle list. This call to **wind_get( )** uses the constant WF_NEXTXYWH.

When all visible rectangles have been examined, function **wind_update( )** is called with a parameter FALSE. This indicates to the AES that the screen has been processed and that it should continue its window processing. Finally, **show_mouse( )** is called to return the mouse to the screen.

The **do_redraw( )** function is designed to handle a single entry point for redrawing a window. However, the redraw function used cannot accept any parameters from **do_redraw( )** function and any returned values are ignored. In most cases, this is not a problem as can be seen in this program and the next. If parameters are needed, they must be used as global variables.

The window-drawing function is located in the application function section of WINDOW1. Function **draw_function( )** draws an ellipse inside a window. For the function to know the window size and shape, it calls function **wind_get( )** to get the work area position and size. First, the window is cleared by using a solid white pattern fill with function **v_bar( )** called to clear the work area. The next step is to select the fill pattern to use. The global variable **cur_window** contains the index of the current window being updated. The four global variables **wind1**, **wind2**, **wind3**, and **wind4** contain the index for window 1, window 2, window 3, and window 4, respectively. These values are set when the window is created. The window currently being updated selects the fill pattern to use. Once the pattern is selected, the color is set to 1 and the ellipse is drawn to fill the entire work area.

The program flow for WINDOW1 handles more events than before but is basically in the same format. Starting with function **main( )**, the initialize GEM access section is the same it has always been. The application-specific routines load the resource file using function **load_resource( )**. Then **init_windows( )** is called to initialize the windows array and the desktop window structure. Function **graf_mouse( )** is called to set the mouse form to the arrow. When an application is initiated from the desktop, GEM converts the mouse form to a bumble bee while the program is loading. Once the program is loaded, GEM passes control to the application but does not return the mouse form

to the arrow image. Therefore, the application should set the mouse form to whatever image is required. The next step in **main( )** is to call function **control( )**. When **control( )** returns to **main( )**, the usual clean-up and exit procedures are followed.

The first thing that function **control( )** does is set the variables **wind1, wind2, wind3,** and **wind4** to the value −1. These four variables hold the indices for the four possible windows. They are global and used by the drawing function. Next **control( )** finds the address of the menu and displays the menu bar. The variable **butstate** is set to TRUE. This is simply used to teach button events. Although button events are not actually used in this program, they are included here to give you an idea as to how buttons are handled. Function **evnt_multi( )** is called to wait for a keyboard, message, or button event. When an event occurs, it may indicate that some kind of window processing needs to be done. Thus, **wind_update( )** is called so that no further window events occur until this event has been processed.

If the event is a message event, the type of message is tested. For menu selection messages, the menu item is tested. Case QUIT means that the Quit option is selected, so variable **end_program** is set to TRUE. Because the program is inside two switch statements and a loop, a single break statement does not exit the loop. Thus the variable **end_program** is used as a flag at the end of the loop. If INFO is selected, the INFOBOX dialog box is drawn. The other menu selections correspond to the opening or closing of a window. If the window is not visible, the menu selection reads "Open Window 1" or whatever window number is appropriate. If the window is visible, the menu selection reads "Close Window 1."

The procedure to open or close a window is the same for each of the four windows. The first test in this case checks if the window has been created. If the window has an index, it must exist in the windows array. If the window's index is −1 (the initialized value), the window must be created. Function **create_window( )** is called specifying the control areas to include and the window title. If the index returned is less than 0, an error must have occurred and the program will exit.

If the window is visible, it should be closed. A call to **close_window( )** closes the window. Then the menu text is changed to read "Open Window 1," or 2, 3, or 4. If the window is not visible (either it was previously closed or has just been created), the window must be opened. Function **open_window( )** opens the appropriate window, and the menu text is changed to indicate that the window can be closed. At the end of the menu selection switch statement, the menu title is reset and the title is redrawn.

The next message case, WM_REDRAW, is for a redraw event. The

redraw event indicates that part of a window work area must be redrawn due to some user action. The message buffer contains the following information. Element 3 contains the window handle to be redrawn. Element 4 contains the x position of the upper left corner of the redraw area. Element 5 contains the y position of the upper left corner of the redraw area. Elements 6 and 7 contain the width and height of the redraw area. All coordinates are measured in screen coordinates.

To handle a redraw event, the program must first get the index of the window being processed. Variable **cur_window** is set to the index returned by **find_window( )**. If **cur_window** is less than 0, the window is not listed in the **windows** array and the redraw process should not continue. If the current window is found, the **do_redraw( )** function is called. The parameters to the **do_redraw( )** function include the address of the window drawing function and the position and size of the redraw area.

Messages WM_NEWTOP and WMTOPPED indicate that a window belonging to this application has become the new top window. The current window is located and the **wind_set( )** function is called with parameter WF_TOP to set the window as the top window. The **wind_set( )** call causes the AES to issue a redraw command for the new top window. Here, as in all of the messages associated with the window, element 3 of the message buffer contains the handle of the window to be acted upon.

Case WM_CLOSED indicates that the close box has been pressed. The index for the window to be closed is located and **close_window( )** is called. Because the window has been closed, the menu entry must be changed to allow the user to reopen this particular window. By testing the current window index against the values in **wind1, wind2, wind3,** and **wind4,** the program determines which menu entry needs to be changed.

Case WM_FULLED is a message sent when the full box is pressed. The **cur_window** value is set and the window size is checked. If the window is already at its full size, the program gets the previous window size and changes the window to that size. A call to **wind_get( )** with parameter WF_PREVXYWH gets the previous window size. Function **wind_set( )** is called with parameter WF_CURRXYWH to change the window size. If the window is in its regular size, the program must make the window its full size. This is done by calling **wind_get( )** with parameter WF_FULLXYWH to get the position and size of a full-size window. The current window size is set to the full-window size by calling **wind_set( )** with the parameter WF_CURRXYWH. The full-size field in the window structure is then toggled using the ^= operator. This operator does a bit-wise XOR between the current value of field full-size and TRUE. If full-size is TRUE, TRUE XOR TRUE yields

FALSE so the value of full-size is toggled. If full-size is FALSE, FALSE XOR TRUE yields TRUE, which also toggles full-size.

Cases WM_ARROWED, WM_HSLID, and WM_VSLID are used for the arrow and slide bars. However, this program does not provide the sliders or arrows. These messages should not occur in this program so they are ignored.

Cases WM_SIZED and WM_MOVED indicate that the window has been resized or moved. Once the current window has been found, elements 6 and 7 contain the requested width and height of the window. These dimensions are tested against the minimum values. Function **wind_set( )** is then called to set the current position, width, and height of the window. This is the last window message event.

The next portion of **control( )** handles a keyboard event. This program has no keyboard events that it should accept so all keyboard events are ignored. The QUIT_KEY case has been commented out so that it no longer functions as it did in MENU2. You can use this case as a model for any other keyboard events your program needs to process. Remember that for menu shortcuts your application must change the menu title state to SELECTED during processing and NORMAL when processing is complete.

The last event handled is a button event. In WINDOW1, the button handler is rather primitive. If a button down event occurs, the program waits for a button up event. When the button up event occurs, the program goes back to waiting for a button down event. This part of the program is included just for demonstration purposes.

The last part of **control( )** checks the **end_program** flag. If **end_program** is TRUE (that is, the end of the program has been requested), all active windows are deleted. If a window has a handle (meaning that it is still active), **del_window( )** is called for the index of that window. Finally, **wind_update( )** is called with parameter FALSE to indicate that window updating may proceed. If **end_program** is not TRUE, the while loop repeats to wait for another event.

Using WINDOW1

WINDOW1 lets you play with various window events. The Windows menu allows you to open or close any one of four windows. The windows can be opened and closed in any order. If you have windows open on the screen, you can close them using either the close boxes or the menu. If there are open windows on the screen and you press the Quit option, the windows are automatically closed before the program exits. Try using the full box, size box, and close box. Try using the move bar to move the windows around.

One interesting experiment is to open window 1 and window 2. Size each window down to about one-quarter of the work area. Place

window 1 slightly below the top of the work area and slightly in from the edge. Move window 2 so that it covers the lower right quarter of window 1. Now move window 2 so that it uncovers window 1. The uncovered portion of window 1 is then redrawn. Note that the area that is uncovered is redrawn with the correct pattern, but that the pattern does not line up with the rest of the window contents. This demonstrates another problem with having *word-aligned* patterns on the screen. Moving a window copies the content of the work area. In this experiment, the content of the work area is copied so that the pattern is no longer word-aligned. To avoid this problem, you can force the window to be redrawn when it is moved or not allow the window to move if it contains a pattern.

Another condition to test is resizing the window. When a window is resized to a smaller width and height, the content of the work area is not redrawn. Only when a window is resized to a larger work area is a redraw message issued.

Look over WINDOW1. WINDOW2 adds a few more control areas and shows you how to handle the data display in the work area. As for WINDOW1, try adding a few enhancements. One would be to change **open_window( )** to allow the application to specify an initial open size. Another would be to specify the full size of the window rather than defaulting to the desktop work area. When you have made some changes to WINDOW1, move on to WINDOW2.

Program WINDOW2

Program WINDOW2 is a text file viewing program. The user opens a text file. This file is loaded into memory and displayed through the window. By using the sliders and scroll bars, the user can view any portion of the file. Closing the window clears the file from memory and allows the user to open another file.

This program combines the operation of program LISTER and program WINDOW1. Program LISTER provides some of the file-accessing routines. Program WINDOW1 supplies the outline for a window-based application. The first thing needed before continuing is a resource file.

Program WINDOW2 Resource File

WINDOW1 uses two object trees (see Figure 12-6): one for the dialog box and one for the menu. The dialog box is named INFOBOX. Set the menu entries as shown in the figure. Again, name the first entry in the Desk menu as INFO. The File menu contains two entries, Open and Quit. Entry Open is named OPENFILE and entry Quit is named QUIT.

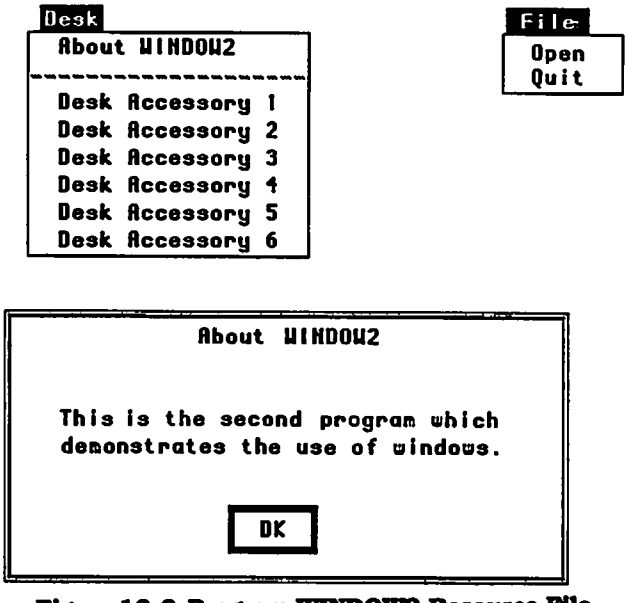


Figure 12-6 Program WINDOW2 Resource File

WINDOW2 Layout

The basic layout of program WINDOW2 is very similar to WINDOW1. The application-specific data section is somewhat different. Take a look at this section as shown in Listing 12-2.

Listing 12-2 Program WINDOW2

```
/*********************************************************************
     WINDOW2.C Sample window application #2

     This program demonstrates the use of the slider and arrows
     on a window.
*********************************************************************/

/*************************************************
     System Header Files & Constants
*************************************************/

#include  <stdio.h>                /* Standard IO */
#include  <osbind.h>               /* GEMDOS routines */
#include  <gemdefs.h>              /* GEM AES */
#include  <obdefs.h>               /* GEM constants */
#include  <errno.h>                /* errno declaration */

#define   FALSE     0
#define   TRUE      !FALSE
```

Listing 12-2 (continued)

```
/*************************************************
     GEM Application Overhead
*************************************************/

/* Declare global arrays for VDI. */
typedef   int  WORD;                /* WORD is 16 bits */
WORD      contrl[12],               /* VDI control array */
          intowt[128], intin[128],  /* VDI input arrays */
          ptsin[128], ptsout[128];  /* VDI output arrays */

WORD      screen_vhandle,           /* virtual screen workstation */
          screen_phandle,           /* physical screen workstation */
          screen_rez,               /* screen resolution 0,1, or 2 */
          color_screen,             /* flag if color monitor */
          x_max,                    /* max x screen coord */
          y_max,                    /* max y screen coord */
          m_hidden = FALSE;         /* mouse visibility status */

WORD      gr_wchar, gr_hchar,       /* values for system sizes */
          gr_wbox, gr_hbox;
                                    /* set min window size */
#define   WMIN_WIDTH     (2*gr_wbox)
#define   WMIN_HEIGHT    (3*gr_hbox)

WORD      ap_id;                    /* application ID */

WORD      num_windows;              /* number of windows open */
WORD      cur_window;               /* index for current window */

typedef   struct wind_type {        /* window record */
     WORD        handle;            /* AES window handle */
     WORD        type;              /* window attribute bits */
     WORD        visible;           /* flag if visible */
     WORD        fullsize;          /* flag if full size */
} WINDOW;

#define   MAX_WINDOW     4          /* allow for 4 windows */
WINDOW    windows[MAX_WINDOW];      /* array of window records */
WINDOW    desktop;                  /* desktop record */
WORD      xdesk, ydesk,             /* desktop area */
          wdesk, hdesk;

/*************************************************
     Application Specific Data
*************************************************/

#include  "window2.h"               /* resource header file */

WORD      fileuin;                  /* file window index */
```

Listing 12-2 (continued)

```
WORD        max_column, max_line,     /* # of cols, lines in file */
            vis_column, vis_line,     /* # of cols, line visible in window */
            cur_column, cur_line,     /* start col, line in window */
            lastcur_line, lastcur_column, /* previous values */
            hdelta, vdelta;           /* size of slider movement */

WORD        xchar, ychar;             /* size of char cell in pixels */

#define     MAX_COLUMNS 256           /* hold 256 chars/ line */
#define     MAX_LINES 1000            /* hold up to 1000 lines */
char        *buf_line[MAX_LINES];     /* lines in buffer */
char        *blank = '\0';            /* blank line */

char def_search[32] =                 /* default search path */
            "A:\*.*",
     sel_file[16],                    /* file selected */
     file_name[64];                   /* full file name to open */

/*************************************************
     GEM-related Functions
*************************************************/

WORD open_vwork(phys_handle)
WORD phys_handle;
/*************************************************
Function: This function opens a virtual workstation.
Input:    phys_handle    = physical workstation handle
Output:   Returns handle of workstation.
*************************************************/
{
WORD work_in[11],
            work_out[57],
            new_handle;                   /* handle of workstation */
int         i;

     for (i = 0; i < 10; i++)             /* set for default values */
            work_in[i] = 1;
     work_in[10] = 2;                     /* use raster coords */
     new_handle = phys_handle;            /* use currently open wkstation */
     v_opnvwk(work_in, &new_handle, work_out);
     return(new_handle);
}

set_screen_attr()
/*************************************************
Function: Set global values about screen.
Input:    None. Uses screen_vhandle.
Output:   Sets x_max, y_max, color_screen, and screen_rez.
*************************************************/
{
WORD work_out[57];
```

Listing 12-2 (continued)

```
     vq_extnd(screen_vhandle, 0, work_out);
     x_max = work_out[0];
     y_max = work_out[1];
     screen_rez = Getrez();          /* 0 = low, 1 = med, 2 = high */
     color_screen = (screen_rez < 2);    /* mono 2, color 0 or 1 */
}

hide_mouse()
/*************************************************
Function: Make mouse invisible if currently visible.
Input:    None. Uses variable m_hidden.
Output:   Sets m_hidden to TRUE.
*************************************************/
{
     if (!m_hidden)
     {
          graf_mouse(M_OFF, 0x0L);
          m_hidden = TRUE;
     }
}

show_mouse()
/*************************************************
Function: Make mouse visible if currently invisible.
Input:    None. Uses m_hidden.
Output:   None. Sets m_hidden to FALSE.
*************************************************/
{
     if (m_hidden)
     {
          graf_mouse(M_ON, 0x0L);
          m_hidden = FALSE;
     }
}

load_resource(rfile)
char *rfile;
/*************************************************
Function: Load resource file.
Input:    rfile     = string with resource file name.
Output:   Returns TRUE if file loaded, else FALSE.
*************************************************/
{
char temp[128];

     if (!rsrc_load(rfile))          /* error loading file */
     {                               /* set alert format */
          sprintf(temp, "[0][Cannot load file %s |Exiting ...] [OK]", rfile);
          form_alert(1, temp);       /* show alert box */
          return(FALSE);
     }
```

Listing 12-2 (continued)

```
    return(TRUE);
}

do_dialog(box_index)
WORD box_index;
/*************************************************
Function: Display a dialog box centered on the screen.
Input:    box_addr  = index of dialog box
Output:   Returns index of object used for exit.
*************************************************/
{
WORD xbox, ybox, hbox, wbox;
WORD smallx, smally, smallw, smallh;
WORD exit_object;
OBJECT    *box_addr;

/* get address of box */
    rsrc_gaddr(0, box_index, &box_addr);

/* get size and location of a box centered on screen */
    form_center(box_addr, &xbox, &ybox, &wbox, &hbox);
    smallx = xbox + (wbox / 2);
    smally = ybox + (hbox / 2);
    smallw = 0;
    smallh = 0;

/* reserve area on screen for box display */
    form_dial(FMD_START,
         smallx, smally, smallw, smallh,
         xbox, ybox, wbox, hbox);

/* draw an expanding box */
    form_dial(FMD_GROW,
         smallx, smally, smallw, smallh,
         xbox, ybox, wbox, hbox);

/* draw dialog box */
    objc_draw(box_addr, 0, 10, xbox, ybox, wbox, hbox);

/* handle dialog input */
    exit_object = form_do(box_addr, 0);

/* draw a shrinking box */
    form_dial(FMD_SHRINK,
         smallx, smally, smallw, smallh,
         xbox, ybox, wbox, hbox);

/* reserve area on screen for box display */
    form_dial(FMD_FINISH,
         smallx, smally, smallw, smallh,
         xbox, ybox, wbox, hbox);
```

Listing 12-2 (continued)

```
/* reset exit object state to unselected */
     box_addr[exit_object].ob_state = NORMAL;

     return(exit_object);
}

set_clip(x, y, w, h)
WORD x, y, w, h;
/*************************************************
Function: Set the clipping rectangle.
Input:    x    = x coord of clip rect
          y    = y coord of clip rect
          w    = width (in pixels) of clip rect
          h    = height (in pixels) of clip rect
Output:   None. Sets new clipping rectangle.
*************************************************/
{
WORD clip[4];

     clip[0] = x;
     clip[1] = y;
     clip[2] = x + w - 1;
     clip[3] = y + h - 1;
     vs_clip(screen_vhandle, TRUE, clip);
}

init_windows()
/*************************************************
Function: Initialize global window values.
Input:    None.
Output:   None. Sets global window values.
*************************************************/
{
int  i;

     num_windows = 0;              /* no windows open */
     for (i = 0; i < MAX_WINDOW; i++)
          windows[i].handle = -1;  /* set records to unused */
     desktop.handle = 0;           /* desktop is always 0 */

                                   /* get desktop work area */
     wind_get(desktop.handle, WF_WORKXYWH,
          &xdesk, &ydesk, &wdesk, &hdesk);
     return;
}

create_window(newkind, wtitle)
WORD      newkind;
char      *wtitle;
/*************************************************
Function: Open a new window.
```

Listing 12-2 (continued)

```
Input:    newkind   = window attributes to include
          wtitle    = string for window title
Output:   Returns index if window created, else -1
Notes:    Info line, and slider values are NOT set
          by this function. Full window size is set to
          desktop work area.
*************************************************/
{
WORD new;

/* check if windows are available */
     if (num_windows >= MAX_WINDOW)      /* max windows in use */
     {
          form_alert(1, "[0][Maximum number of windows reached.][OK]");
          return(-1);
     }

/* find window record to use */
     for(new = 0; new < MAX_WINDOW; new++)
          if (windows[new].handle < 0)   /* record found */
               break;
     if (new >= MAX_WINDOW)              /* no records available */
     {
          form_alert(1, "[0][No window records found available.][OK]");
          return(-1);
     }

/* create window for AES */
     windows[new].handle =
          wind_create(newkind, xdesk, ydesk, wdesk, hdesk);

     if (windows[new].handle < 0)        /* AES could not make window */
     {
          form_alert(1, "[0][AES error opening a window.|Cannot continue.][OK]")
          return(-1);
     }

/* fill window record */
     windows[new].type = newkind;
     windows[new].fullsize = FALSE;
     windows[new].visible = FALSE;

/* set window title */
     wind_set(windows[new].handle, WF_NAME, wtitle, 0, 0);

     num_windows++;                  /* add window to count */
     return(new);                    /* return index */
}

open_window(wi_index)
/*************************************************
Function: Make a window visible.
```

Listing 12-2 (continued)

```
Input:    wi_index  = index to window
Output:   None. Sets work area and visibility flag in window record.
************************************************/
{
WORD xsize, ysize, wsize, hsize;

/* check if already open */
     if (windows[wi_index].visible)
          return;

/* get current window size */
     wind_get(windows[wi_index].handle, WF_PREVXYWH,
          &xsize, &ysize, &wsize, &hsize);

/* if no size, then must be first time opened */
     if ( !(xsize || ysize || wsize || hsize) )
     {
          xsize = xdesk;           /* default to desktop work area */
          ysize = ydesk;
          wsize = wdesk;
          hsize = hdesk;
     }

/* open window */
     hide_mouse();
     graf_growbox((xsize + wsize/2), (ysize + hsize/2),
          gr_wbox, gr_hbox, xsize, ysize, wsize, hsize);
     wind_open(windows[wi_index].handle, xsize, ysize, wsize, hsize);
     windows[wi_index].visible = TRUE;flag window is open */
     show_mouse();
}

close_window(wi_index)
/************************************************
Function: Close a window.
Input:    wi_index  = index to window to close
Output:   None. Closes window and set visibility flag.
************************************************/
{
WORD xsize, ysize, wsize, hsize;

/* check if already closed */
     if (!windows[wi_index].visible)
          return;

/* get current window area */
     wind_get(windows[wi_index].handle, WF_CURRXYWH,
          &xsize, &ysize, &wsize, &hsize);

/* close window */
     hide_mouse();
     wind_close(windows[wi_index].handle);
```

Listing 12-2 (continued)

```
     graf_shrinkbox((xsize + wsize/2), (ysize + hsize/2),
          gr_wbox, gr_hbox, xsize, ysize, wsize, hsize);
     windows[wi_index].visible = FALSE; /* flag window as closed */
     show_mouse();
}

del_window(wi_index)
WORD wi_index;
/*************************************************
Function: Delete window from AES
Input:    wi_index  = index of window to delete
Output:   None. Window deleted from AES and windows.
*************************************************/
{
     if (windows[wi_index].visible)/* still on screen */
          close_window(wi_index);  /* so close it */

/* delete from AES */
     wind_delete(windows[wi_index].handle);
/* set record as available */
     windows[wi_index].handle = -1;
/* remove window from count */
     nwm_windows--;
}

find_window(wi_handle)
WORD wi_handle;
/*************************************************
Function: Find index for window record with given handle.
Input:    wi_handle = handle of window to search for
Output:   Returns index or -1 if not found.
*************************************************/
{
register  int  i;

     for (i = 0; i < MAX_WINDOW; i++)
          if (windows[i].handle == wi_handle)
               return(i);
     return(-1);
}

do_redraw(wi_index, wi_redraw, x, y, w, h)
WORD wi_index,
     (*wi_redraw)(),
     x, y, w, h;
/*************************************************
Function: Redraw all clipping rectangles.
Input:    wi_index  = index to window being updated
          wi_redraw = address of function used to draw window
          x, y      = redraw area X,Y coord
          w, h      = redraw area width and height
```

Listing 12-2 (continued)

```
Output:   None. Screen is updated.
Notes:    wi_redraw cannot use any parameters or return any values.
          You may create a do_redraw function for each window,
          or pass the parameters through global variables.
*************************************************/
{
GRECT     redraw,                    /* redrau area */
          clip;                      /* current clip area */

     hide_mouse();
     wind_update(TRUE);              /* freeze window status */
     redraw.g_x = x;                 /* set redraw area */
     redraw.g_y = y;
     redraw.g_w = w;
     redraw.g_h = h;

/* get first clip rectangle */
     wind_get(windows[wi_index].handle, WF_FIRSTXYWH,
          &clip.g_x, &clip.g_y, &clip.g_w, &clip.g_h);

/* begin redraw loop until no more clip rectangles */
     while (clip.g_w && clip.g_h)
     {
          if (rc_intersect(&redraw, &clip))
          {
               set_clip(clip.g_x, clip.g_y, clip.g_w, clip.g_h);
               (*wi_redraw)();     /* particular redraw function */
          }
          wind_get(windows[wi_index].handle, WF_NEXTXYWH,
               &clip.g_x, &clip.g_y, &clip.g_w, &clip.g_h);
     }
     wind_update(FALSE);             /* screen is ready */
     show_mouse();
}

/*************************************************
     Application Functions
*************************************************/

get_file()
/*************************************************
Function: Get a filename using file selector.
Input:    None. Uses def_search[] and sel_file[].
Output:   Exit button used in fsel dialog box.
          Sets default file search, file selected,
          and file_name[] to contain the full filename.
*************************************************/
{
WORD exit_button;
char *temp;
int  i;
```

Listing 12-2 (continued)

```
     fsel_input(def_search, sel_file, &exit_button);
     if (!exit_button)                    /* cancelled */
          return(exit_button);

     strcpy(file_name, dsf_search);       /* set path */
     i = strlen(file_name);
     temp = file_name + (i - 1);          /* point to last character */
     while ( (*temp != '\\')              /* search for path end */
               && (*temp != ':')          /* drive id */
               && (temp >= file_name) )   /* or start of string */
          temp--;
     temp++;                              /* move to next position */
     strcpy(temp, sel_file);              /* add file name */
     return(exit_button);
}

get_line(s, fp, maxchar)
char s[];
FILE *fp;
WORD maxchar;
/*************************************************
Function: Get a line of text from a file.
Input:    s     = string to store line
          fp    = stream file to receive data from
          maxchar    = max # of characters to read
Output:   Returns number of characters read.
Notes:    File fp must be open. Line does NOT contain newline.
          From K&R p.67.
*************************************************/
{
int  c, i;

     i = 0;
     while (--maxchar > 0 && (c=getc(fp)) != EOF && c != '\n')
          s[i++] = c;
     if (c == '\n')
          s[i++] = c;
     s[i] = '\0';
     return(i);
}

load_file()
/*************************************************
Function: Load a file into the buffer.
Input:    None. Uses file_name[] and buf_line[]
Output:   Returns TRUE if error occurred, else FALSE.
          Sets buf_line[].
*************************************************/
{
FILE *fp;
char line[MAX_COLUMNS], *temp;
int   numchar;
```

Listing 12-2 (continued)

```
/* open file */
     fp = fopen(file_name, "r");
     if (fp == NULL)   /* error opening file */
     {
          form_alert(0,"[0][Cannot open file requested.][OK]");
          return(TRUE);
     }

/* read a line at a time (from K&R p.107) */
     max_line = 0;
     max_column = 0;
     while ((numchar = get_line(line, fp, MAX_COLUMNS)) > 0)
     {
          if (max_line >= MAX_LINES)
          {
               form_alert(0,"[0][Maximum number of lines read.|No more data will be read.][OK]");
               return(FALSE);
          }
          else if ( (temp = malloc(nwmchar)) == NULL)
          {
               form_alert(0,"[0][Out of memory.|No more data will be read.][OK]");
               return(FALSE);
          }
          else
          {
               strcpy(temp, line);
               buf_line[max_line++] = temp;
          }
          if (numchar > max_column) /* check for max uidth */
               max_column = numchar;
     }
     return(FALSE);
}

set_sliders(wi_index)
/*************************************************
Function: Set slider size and position.
Input:    None. Uses global file sizes.
Output:   None. Sets new slider size and position.
*************************************************/
{
WORD xwork, ywork, wwork, hwork;   /* window work area */
WORD hsize, vsize, hpos, vpos;     /* slider size and position */

     wind_get(windows[wi_index].handle, WF_WORKXYWH,
          &xwork, &ywork, &wwork, &hwork);
     vis_line = hwork / ychar;
     vis_column = wwork / xchar;
                                   /* find size of slider */
     hsize = ((long)vis_column * 1000L) / (long)max_column;
     vsize = ((long)vis_line * 1000L) / (long)max_line;
     if (vsize > 1000) vsize = 1000;
```

Listing 12-2 (continued)

```
     if (hsize > 1000) hsize = 1000;
     hpos = ((long)cur_column * 1000L) /
                    (long)(max_column - vis_column);
     if (hpos < 0) hpos = 0;
     vpos = ((long)cur_line * 1000L) /
                    (long)(max_line - vis_line);
     if (vpos < 0) vpos = 0;
     wind_set(windows[wi_index].handle, WF_HSLIDE, hpos);
     wind_set(windows[wi_index].handle, WF_HSLSIZE, hsize);
     wind_set(windows[wi_index].handle, WF_VSLIDE, vpos);
     wind_set(windows[wi_index].handle, WF_VSLSIZE, vsize);
     return;
}

draw_function()
/*************************************************
Function: Draw in current window.
Input:    None. cur_window is index to current window to draw in.
Output:   None. Draws in window.
*************************************************/
{
WORD line_pos, col_pos;            /* line & column positions */
WORD xwork, ywork, wwork, hwork;   /* window area */
WORD box[4];                       /* clear screen array */
char *temp;                        /* max chars/line */
WORD xpos, ypos;                   /* line position on screen */
WORD hal, val;                     /* dummy return values */

/* get window position on screen */
     wind_get(windows[cur_window].handle, WF_WORKXYWH,
          &xwork, &ywork, &wwork, &hwork);

/* clear screen */
     vsf_interior(screen_vhandle, 2);
     vsf_style(screen_vhandle, 8);
     vsf_color(screen_vhandle, 0);
     box[0] = xwork;
     box[1] = ywork;
     box[2] = xwork + wwork - 1;
     box[3] = ywork + hwork - 1;
     v_bar(screen_vhandle, box);
     vst_alignment(screen_vhandle, 0, 5, &hal, &val);

/* output */
     xpos = xwork;
     ypos = ywork;
     for (line_pos = 0; line_pos <= vis_line; line_pos++)
     {
          if (cur_line+line_pos < max_line)  /* check if line in range */
               temp = buf_line[cur_line+line_pos];
          else
               temp = blank;
```

Listing 12-2 (continued)

```
            if (strlen(temp) <= cur_column)     /* nothing will be seen */
                temp += strlen(temp);
            else
                temp += cur_column;             /* adjust to proper start */
            v_justified(screen_vhandle, xpos, ypos,
                temp, wwork, FALSE, FALSE);
            ypos += ychar;
        }

        return;
}


control()
/*************************************************
Function: Master control function.
Input:    None. Program initialization must be done before
          entering this function.
Output:   None. Returns for normal program termination.
*************************************************/
{
OBJECT    *menu_addr;               /* address for menu */
WORD      end_program = FALSE;      /* exit flag */
WORD      mevx, mevy,               /* evnt_multi parameters */
          butstate,
          mevbut,
          keystate,
          keycode,
          mbreturn,
          msg_buf[8];               /* message buffer */
WORD      event,                    /* evnt_multi result */
menu_index;                         /* keyboard menu title selected */
WORD      draw_function();          /* screen drawing function */

/* get address of menu */
     rsrc_gaddr(0, MAINMENU, &menu_addr);

/* display menu bar */
     menu_bar(menu_addr, TRUE);

/* initial button status to wait for */
     butstate = TRUE;               /* button down */

/* wait for a message indicating a menu selection */
do                                  /* continue loop until quit */
{
     event = evnt_multi( (MU_KEYBD | MU_MESAG | MU_BUTTON),
          0,                        /* # mouse clicks */
          0,                        /* mouse buttons of interest */
          butstate,                 /* button state */
          0,                        /* first rectangle flags */
          0, 0,                     /* x,y of 1st rectangle */
```

Listing 12-2 (continued)

```
        0, 0,                   /* height, widht of 1st rect */
        0,                      /* second rectand flags */
        0, 0,                   /* x,y of 2nd rect */
        0, 0,                   /* w,h of 2nd rect */
        msg_buf,                /* message buffer */
        0, 0,                   /* low, high words for timer */
        &mevx, &mevy,           /* x,y of mouse event */
        &mevbut,                /* button state at event */
        &keystate,              /* status of keyboard at event */
        &keycode,               /* keyboard code for key pressed */
        &mbreturn);             /* # times mouse key enter state */

    wind_update(TRUE);          /* hold window processing */

    if (event & MU_MESAG)
    {
     switch(msg_buf[0])
     {
     case MN_SELECTED:          /* menu chosen */
        switch(msg_buf[4])
        {
        case QUIT:              /* exit program */
             end_program = TRUE; /* set exit flag */
             break;

        case INFO:              /* display program info */
             do_dialog(INFOBOX);
             break;

        case OPENFILE:          /* open a file */
             if (!get_file())   /* get a file name */
                  break;        /* cancelled */
             if (load_file())   /* get data */
                  break;        /* error */
             if (max_line == 0 || max_column == 0)
                  break;        /* bad data */
             filewin = create_window(
                  (NAME|CLOSER|FULLER|MOVER|SIZER|
                  UPARROW|DNARROW|VSLIDE|
                  LFARROW|RTARROW|HSLIDE),
                  file_name);
             if (filewin < 0)   /* error creating window */
             {
                  end_program = TRUE;
                  break;
             }
             open_window(filewin);
             menu_ienable(menu_addr, OPENFILE, FALSE);
             lastcur_line = cur_line = 0;
             lastcur_column = cur_column = 0;
             cur_window = filewin;
             set_sliders(filewin);
             break;
```

Listing 12-2 (continued)

```
        default:
             break;
        }
                                /* reset menu title */
        menu_tnormal(menu_addr, msg_buf[3], 1);
        menu_bar(menu_addr, TRUE);
        break;                  /* end MN_SELECTED */

    case WM_REDRAW:             /* redraw windows */
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;             /* no window listed */
        do_redraw(cur_window, draw_function,
             msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7]);
        break;

    case WM_NEWTOP:             /* new window is on top */
    case WM_TOPPED:
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;             /* no window listed */
        wind_set(windows[cur_window].handle, WF_TOP, 0, 0, 0, 0);
        break;

    case WM_CLOSED:             /* close box pressed */
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;             /* no window listed */
        del_window(cur_window);
        msnu_ienable(menu_addr, OPENFILE, TRUE);
        break;

    case WM_FULLED:             /* full box pressed */
        if ( (cur_window = find_window(msg_buf[3])) < 0)
             break;             /* no window listed */
        if (windows[cur_window].fullsize)
        {                       /* full to regular size */
        WORD newx, newy, neww, newh;
                                /* get previous window size */
             wind_get(windows[cur_window].handle, WF_PREVXYWH,
                  &newx, &newy, &neww, &newh);
             wind_set(windows[cur_window].handle, WF_CURRXYWH,
                  newx, newy, neww, newh);
             set_sliders(cur_window);
        }
        else
        {                       /* regular to full size */
        WORD xfull, yfull, wfull, hfull;
                                /* get full window size */
             wind_get(windows[cur_window].handle, WF_FULLXYWH,
                  &xfull, &yfull, &wfull, &hfull);
             wind_set(windows[cur_window].handle, WF_CURRXYWH,
                  xfull, yfull, wfull, hfull);
             set_sliders(cur_window);
        }
        windows[cur_window].fullsize ^= TRUE;
        break;
```

Listing 12-2 (continued)

```
case WM_ARROWED:
     if ( (cur_window = find_window(msg_buf[3])) < 0)
          break;              /* no window listed */
     switch(msg_buf[4])
     {
     case 0:                  /* page up */
          cur_line -= vis_line;
          break;
     case 1:                  /* page down */
          cur_line += vis_line;
          break;
     case 2:                  /* row up */
          cur_line--;
          break;
     case 3:                  /* row down */
          cur_line++;
          break;
     case 4:                  /* page left */
          cur_column -= vis_column;
          break;
     case 5:                  /* page right */
          cur_column += vis_column;
          break;
     case 6:                  /* column left */
          cur_column--;
          break;
     case 7:                  /* column right */
          cur_column++;
          break;
     default:
          break;
     }
                              /* check if values are in range */
     if (cur_line >=  max_line-vis_line )
               cur_line = max_line - vis_line;
     if (cur_column >= max_column-vis_column)
               cur_column = max_column - vis_column;
     if (cur_line < 0) cur_line = 0;
     if (cur_column < 0) cur_column = 0;
     if (lastcur_line != cur_line || lastcur_column != cur_column)
     {
          set_sliders(cur_window);
          lastcur_line = cur_line;
          lastcur_column = cur_column;
          msg_buf[0] = WM_REDRAW;
          msg_buf[1] = ap_id;
          msg_buf[2] = 0;
          msg_buf[3] = windows[cur_window].handle;
          wind_get(windows[cur_window].handle, WF_WORKXYWH,
               &msg_buf[4], &msg_buf[5], &msg_buf[6], &msg_buf[7]);
          appl_write(ap_id, 16, msg_buf);
     }
     break;
```

Listing 12-2 (continued)

```
case WM_HSLID:
     if ( (cur_window = find_window(msg_buf[3])) < 0)
          break;              /* no window listed */
     cur_column =
          ((long)(max_column - vis_column) * (long)msg_buf[4]) /
               1000L;
     if (cur_column < 0) cur_column = 0;
     if (cur_column >= max_column)
     cur_column = max_column - 1;
     if (lastcur_column != cur_column)
     {
     WORD xtemp, ytemp, wtemp, htemp;
          set_sliders(cur_window);
          lastcur_line = cur_line;
          wind_get(windows[cur_window].handle, WF_WORKXYWH,
            &xtemp, &ytemp, &wtemp, &htemp);
          do_redraw(cur_window, draw_function,
            xtemp, ytemp, wtemp, htemp);
     }
     break;


case WM_VSLID:
     if ( (cur_window = find_window(msg_buf[3])) < 0)
          break;              /* no window listed */
     cur_line =
               ((long)(max_line - vis_line) * (long)msg_buf[4]) /
                    1000L;
     if (cur_line < 0) cur_line = 0;
     if (cur_line >= max_line)
               cur_line = max_line - 1;
     if (lastcur_line != cur_line)
     {
     WORD xtemp, ytemp, wtemp, htemp;
          set_sliders(cur_window);
          lastcur_line = cur_line;
          wind_get(windows[cur_window].handle, WF_WORKXYH,
            &xtemp, &ytemp, &wtemp, &htemp);
          do_redraw(cur_window, draw_function,
            xtemp, ytemp, wtemp, htemp);
     }
     break;


case WM_SIZED:                /* window resized or moved */
case WM_MOVED:
     if ( (cur_window = find_window(msg_buf[3])) < 0)
          break;              /* no window listed */
     if (msg_buf[6] < WMIN_WIDTH) msg_buf[6] = WMIN_WIDTH;
     if (msg_buf[7] < WMIN_HEIGHT) msg_buf[7] = WMIN_HEIGHT;
     wind_set(windows[cur_window].handle, WF_CURRXYWH,
          msg_buf[4], msg_buf[5], msg_buf[6], msg_buf[7]);
     set_sliders(cur_window);
     break;
```

Listing 12-2 (continued)

```
        default:
           break;
       }                              /* end message switch */
      }                               /* end message handler */

      if (event & MU_KEYBD)
      {
       switch(keycode)
       {
/*      case QUIT_KEY:
           menu_index = FILE;
           menu_tnormal(menu_addr, menu_index, 0);
           end_program = TRUE;
           break;
*/
        default:
           break;
       }
       menu_tnormal(menu_addr, menu_index, 1);
       menu_bar(menu_addr, TRUE);
      }                               /* end keyboard handler */

      if (event & MU_BUTTON)          /* button handler */
      {
           butstate = !(butstate);
      }                               /* end button handler */

      if (end_program)                /* close open windows */
      {                               /* and delete all windows */
      int  i;

           for (i = 0; i < MAX_WINDOW; i++)
                if (windows[i].handle >= 0)
                     del_window(i);
      }

      wind_update(FALSE);             /* resume processing */

 } while (!end_program);              /* program control loop */

 return;
 }                                    /* end function */

 /***********************************************
      Main Program
 ***********************************************/

 main()
 {
 int  i, j;
 WORD attr[10];
```

Listing 12-2 (continued)

```
/*************************************************
     Initialize GEM Rocess
*************************************************/

     ap_id = appl_init();          /* Initialize AES routines */
     if (ap_id < 0)                /* no calls can be made to AES */
     {                             /* use GEMDOS */
          Cconws("***> Initialization Error. <***\n");
          Cconws("Press any key to continue.\n");
          Crawcin();
          exit(-1);                /* set exit value to show error */
     }
     screen_phandle =              /* Get handle for screen */
          graf_handle(&gr_wchar, &gr_hchar, &gr_wbox, &gr_hbox);
     screen_vhandle = open_vwork(screen_phandle);
     set_screen_attr();            /* Get screen attributes */

/*************************************************
     Application Specific Routines
*************************************************/

     if (!load_resource("WINDOW2.RSC")) /* no resource file loaded */
          exit(1);
     init_windows();
     graf_mouse(ARROW, 0x0L);

     vqt_attributes(screen_vhandle, attr);
     xchar = attr[8];
     ychar = attr[9];

     control();

/*************************************************
     Program Clean-up and Exit
*************************************************/

     rsrc_free();
     v_clsvwk(screen_vhandle);     /* close workstation */
     appl_exit();                  /* end program */
}
/*************************************************/
/*************/
```

First comes the header file WINDOW2.H for the resources followed by the variable **filewin**. This variable contains the window index for the file being used. The next set of variables is used to determine the size of the file and the size of the display. The **max_column** and **max_line** variables have the maximum number of columns and maximum number of lines contained in the file. Variables **vis_column**

and **vis_line** count the number of visible columns and visible lines in the window. Because the window can change size, it is important to know how much data is visible. Variables **cur_column** and **cur_line** keep track of the starting column number and line number currently shown in the window. As the user moves through the data in the file, knowing the starting position of the data being displayed is necessary. Variables **lastcur_line** and **lastcur_column** hold the previous values of **cur_column** and **cur_line.** The **xchar** and **ychar** variables are used to store the width and height, respectively, of the character cell in pixels. Constants MAX_COLUMNS and MAX_LINES define the maximum allowable number of characters per line and of lines in the file. Array **buf_line** is an array of pointers to strings. As the lines are read in, the **buf_line** array keeps track of them. The string **blank** is simply a blank line. Finally, there are the default search path, selected file, and file name variables as used in program LISTER.

Using WINDOW1 as an outline, the GEM-related functions remain the same. Moving to the application functions of WINDOW2, the first function encountered is function **get_file( )**. This function is exactly the same as in program LISTER. The next function, **get_line( )**, is a more generalized version of the **getline( )** function by Kernighan & Ritchie (Kernighan, Brian & Ritchie, Dennis, *The C Programming Language*, Prentice-Hall, 1978, p. 67). The **get_line( )** function presented here reads a line from a file. A line consists of a series of bytes terminated by a newline character ('\n'). The bytes, read from file **fp** in the parameter list, are placed into string parameter **s.** If the number of bytes reaches the value of parameter maxchar, reading stops and the function returns. So, either a newline or the maximum number of characters will terminate a line. This function returns the number of characters read for this line.

Function **load_file( )** opens the file named in **file_name** and reads the contents of the file into memory using **get_line( )**. If an error occurs during opening or reading the file, an alert box is given and the function returns a FALSE value. While reading the file, **load_file( )** sets the maximum line length in **max_column** and the total number of lines read in **max_line**.

Function **set_sliders( )** sets the slider size and position in the scroll bar. The size and position of the slider are measured like the **graf_slidebox( )** function. The size of the slider can range from 0 for a slider of no size, to 1000 for a slider covering the entire scroll bar. The first bit of information **set_slider( )** needs is the size of the window display in characters. Function **wind_get( )** retrieves the size of the work area in pixels. The size of the file is measured in characters so that the size of the work area must be converted from pixels to characters. The number of lines visible in the work area (**vis_line**) is equal to the height of the work area in pixels (**hwork**)

divided by the number of pixels per character height (**ychar**). Similarly, the width of the work area in characters (**vis_column**) is found by dividing the width of the work area (**wwork**) by the number of pixels per character width (**xchar**). The ratio of the slider size to the size of the scroll bar (1000) is equal to the ratio of the size of the window display to the size of the total data to display. Therefore, the ratio for the horizontal slider can be written as this.

$$\frac{\textbf{hsize}}{1000} = \frac{\textbf{vis\_column}}{\textbf{max\_column}}$$

To solve for the relative slider size (**hsize**), the following equation is used in **set_slider( )**.

hsize = (**vis_column** * 1000) / **max_column**

For the vertical slider, the equation is this.

vsize = (**vis_line** * 1000) / **max_line**

Long integers are used to allow the multiplication, which may exceed the range of standard integers, to occur first. Since all values are integers, serious truncation errors may occur if the division occurred first and the results would not be accurate. Once the slider sizes have been calculated, a test ensures that they do not exceed the size of the scroll bar.

In program MOUSE, you saw that the position of a slider in a slide bar using **graf_slidebox( )** is measured at the center of the slider itself. The position of the slider in the window scroll box is with respect to the position of one end of the slider or the other. For WINDOW2, the position of the slider is with respect to the bottom of the slider (see Figure 12-7). For example, when the slider is all the way at the top, it is considered to be at position 0; at the bottom it is considered to be at position 1000.

In function **set_sliders( )**, variables **hpos** and **vpos** are used for the horizontal and vertical positions of the sliders. The position of the slider in the scroll box is relative to the position of the displayed data in the file. The ratio for the horizontal scroll bar is this:

$$\frac{\textbf{hpos}}{1000} = \frac{\textbf{vis\_column}}{\textbf{max\_column} - \textbf{vis\_column}}$$

If you recall from program MOUSE, the actual range in which the slider can be moved is the size of the slide bar minus the size of the

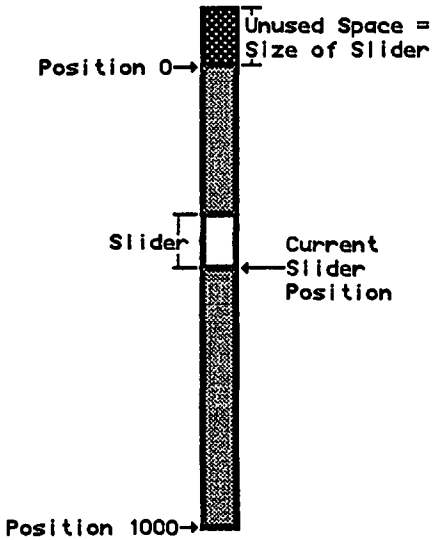


Figure 12-7 Slide Bar Measurements

slider. As shown in the ratio to calculate the size of the slider, the size of the slider is equivalent to the number of visible columns and the size of the scroll bar is equivalent to the total number of columns. Therefore, the entire range of the scroll bar (1000) is equivalent to the total number of columns (the size of the scroll bar) minus the number of visible columns (the size of the slider). In effect, the size of the scroll bar has been reduced by the size of the slider. To calculate the relative horizontal position, **set_slider( )** uses this:

$$\mathbf{hpos} = \frac{(\mathbf{cur\_column} * 1000)}{(\mathbf{max\_column} - \mathbf{vis\_column})}$$

The vertical position is calculated using this:

$$\mathbf{vpos} = \frac{(\mathbf{cur\_line} * 1000)}{(\mathbf{max\_line} - \mathbf{vis\_line})}$$

In function **set_slider( )**, the long integer values are used for the same reason mentioned above.

After the slider sizes and positions are calculated, they need to be set in the window. Function **wind_set( )** sets each value.

The **draw_function( )** routine draws the visible range of data in the current window. The routine first gets the position and size of the work area on the screen. The window is cleared with a call to the

v_bar( ) VDI function. Variable **line_pos** is used to count the line number in the window. The top line is line 0, and the bottom line has the value **vis_line**. The top line corresponds to the string in the **buf_line** array with index **cur_line**. Thus, each line in the buffer is indexed by **line_pos** plus **cur_line**.

To draw text on the screen, a VDI text output routine must be used for two reasons. First, the AES coordinates measure the upper left corner of rectangles. This corresponds to the top of a character cell. Therefore, the **vst_alignment( )** function is used to set top vertical text alignment. The second reason for using a VDI text output function is that other text output routines do not recognize the clipping rectangle. Because the clipping rectangle is an important aspect of the window redraw process, all window output routines must recognize the clipping rectangle. The VDI text output routines, **v_gtext( )** and **v_justified( )**, both require a starting x and y coordinate. The variable **xpos** and **ypos** keep track of the current line position in screen coordinates.

A for loop is used to output the lines on the screen. Since the number of lines to be output is **vis_line** + 1, the loop control variable, **line_pos**, starts at index 0 and goes to **vis_line**. For each line on the screen, the line index is tested to see if it is less than the maximum number of lines. If the index is valid, the pointer **temp** is set to point to this line in the buffer. If the index is greater than or equal to **max_line**, the index has exceeded the maximum number of lines in the buffer (that is, the window extends past the end of the data). In this case, **temp** is set to point to the blank line defined at the top of the program.

The next step in displaying the line determines the starting column within the string. If the length of the line is less than the starting column, the left edge of the window starts beyond the end of this line so nothing should be visible. Pointer **temp** is moved to the null character at the end of the string so that when the string is output nothing is displayed. If the length of the line is greater than the current starting column, pointer **temp** is moved to start at the proper position in the string. The text is printed using **v_justified( )**. Unjustified text mode is used here. If you want to output justified text, you can make the appropriate changes. Once the line has been displayed, the y position, **ypos**, is incremented by the height of a character cell.

As a note, this particular draw function relies on the activation of the clipping rectangle before the text is drawn because **v_justified( )** does not stop output based upon the parameter for the length of the string (**wwork**). Function **v_justified( )** uses this value as a minimum length of the string. If the string exceeds this length, **v_justified( )** continues to print until the end of the string is reached. By setting the clipping rectangle, **v_justified( )** is prevented from printing beyond the right edge of the screen. Another reason the clipping rectangle

must be set beforehand is that the number of lines printed is one more than **vis_line.** Therefore, the last line in the window exceeds the window height and needs to be clipped. In general, the user would not set the window height to hold an exact number of lines. The program can either not print the partial last line and leave extra space or clip the last line as done here. Another alternative would be to test the size of the work area whenever the window is resized and adjust the new size to an exact number of lines. This same explanation also applies to the width of the work area.

Function **control( )** follows the same structure as the **control( )** function from WINDOW1. In the menu selections, the OPENFILE case opens a disk file and displays it in the window. First, **get_file( )** is called to allow the user to select a file. A FALSE returned value indicates that the user has cancelled the open request and the case is exited. Otherwise, the requested file is opened and loaded through **load_file( )**. If the loading is successful, **max_line** and **max_column** should have values other than 0. If they don't, either the data was not read in properly or the data cannot be read in as text. In either case, the function is cancelled.

With the data in memory, a window must be opened and the data displayed. The **create_window( )** function is used to create the window with all control areas except the information line. These areas are the name box, close box, full box, move box, size box, the up and down arrow, the vertical slider, the left and right arrow, and the horizontal slider. The title of the window is the filename. Variable **file_win** is set to the index of the window used for the file. If the window was created properly, **file_win** has a value of 0 or greater. The window is then opened, and the menu selection to open a file is disabled. The variables **lastcur_line**, **cur_line**, **lastcur_column**, and **cur_column** are set to 0 so that the display in the window starts at the first line and the first column. Variable **cur_window** is set to **file_win** and the sliders are set.

Because a window is created each time a file is open, when a window is closed, it must be deleted from the system; otherwise the program runs out of windows to use. Therefore, the WM_CLOSED case deletes the window instead of just closing it.

Not every window interaction with the user generates a redraw message. Only in cases where displayable area of the window increases is a redraw message issued. For example, when the window becomes the new top window, the work area increases in size or another window is moved to show more of this program's window. Closing a window naturally does not issue a redraw message because the window is no longer visible. Moving a window does not generate a redraw message because the AES is responsible for copying the entire window contents from its first location to its final location.

The arrows, scroll boxes, and sliders do not automatically cause a redraw message to be issued. The AES does not know what data is being represented in the window. It does not know what reaction would be appropriate. For example, it does not know the number of lines the window should move or whether the display is at the end of the data. Therefore, in the case of arrow selection, scroll bar selection, or slider movement, the redraw process must be handled by the application. The redraw process can be handled either by performing the redraw function or issuing a redraw message.

The WM_ARROWED message is used whenever the user interacts with the scroll bar or the arrows. Element 4 of the message buffer contains a value indicating the control area requested (see Table 12-1). The page up, page down, row up, or row down requests correspond to a click above the slider, a click below the slider, or clicking the up or down arrows, respectively. Page left, page right, column left, and column right correspond to the horizontal scroll bar and arrows.

Table 12-1: Message WM_ARROWED Border Selections

Value	*Selection*
0	Page up
1	Page down
2	Row up
3	Row down
4	Page left
5	Page right
6	Column left
7	Column right

Depending upon the request, the appropriate action upon **cur_line** or **cur_column** is taken in each event. If the event is a page up or down, the current line is decreased or increased by the number of visible lines (for example, one page). An up or down arrow causes the current line to be decreased or increased by one line. Similar actions are taken for the horizontal movements.

After the current line or column is changed, the new value must be tested to see if it remains within the range of the data. WINDOW2 does not allow the user to move the window past the end of the file. The tests keep the starting line and column greater than or equal to 0 and within one screen size of the end of the data. Now that the new starting line or column is known, the window must be redrawn to show the new area of data. If the new starting line or column is the

same as the old starting line or column, the user's request causes no changes to the display and the redrawing is skipped.

The WM_ARROWED case sends a message to initiate the redraw process. First the new slider positions are set. Then the current value becomes the previous value. To send a message, the message buffer must be filled with the proper values. Element 0 is the message type, so it is filled with the WM_REDRAW constant value. Element 1 is the ID number of the application sending the message. Since this message fits within the 16 bytes allowed, element 2 is 0. For a redraw message, element 3 must contain the handle of the window to redraw and elements 4, 5, 6, and 7 define the redraw area. Since this is the top window, the entire work area is visible and needs to be redrawn. The function **appl_write( )** sends the message to the AES, which, in turn, routes the message to the appropriate application. In this case, WINDOW2 is writing a message to itself. This technique is shown here to demonstrate the process of sending a message to an application. Generally, an application would not send a message to itself.

Cases WM_HSLID and WM_VSLID respond to the new slider positions requested by the user. In these messages, element 4 of the message buffer contains the new position of the slider. From this new position, a new current line is calculated based upon the equations discussed above. In the equation used in these cases, **msg_buf[4]** represents **hpos** or **vpos** in WM_HSLID or WM_VSLID, respectively. If a new current line or column has been requested, the **do_redraw( )** function is called to redraw the window. Calling the **do_redraw( )** function directly is much more efficient than having the program send a message to itself.

Finally, in function **main( )** all initialization procedures are the same as in WINDOW1. The global variables **xchar** and **ychar** are set with the values returned by the **vqt_attributes( )** function. Then function **control( )** is called.

That about does it for this program. As for testing the program, read in some of the source code files you have created for this book. If you entered program WINDOW2 as shown in Listing 12-2, your source file should have over 1000 lines. If you try to read this source file with the WINDOW2 program, the alert box indicates that the maximum number of lines has been reached. To enhance the program, add another window so the user can open more than one file at a time. To do this, you need to keep track of which window belongs to which file. You might want to create your own data structure to keep track of this relationship.

You have seen and used most of the GEM and Atari functions accessible through the C programming language. A complete operational listing of all functions is given in Appendix A. The appendices also include constant definitions, standard header files, and useful tables.

Enjoy programming your Atari ST. Remember to keep your programs organized and modular. To use WINDOW1 or WINDOW2 as an outline file, simply delete the statements in the sections Application-Specific Data, Application Functions, and Application-Specific Routines in function **main( )**. Study the programs presented in this book and don't be afraid to experiment with them, especially the AES programs. Menus, windows, and the graphic interface change the way a program looks. Full utilization of GEM and Atari routines can give a program professional elegance and sophistication.

APPENDIX A

C Function Reference Guide

This appendix is a reference guide to all functions defined in the Atari system developer's documentation. The function names and parameter lists given here are in accordance with Atari documentation. The *exact* syntax of a function name (for example, upper- or lower-case letters) may vary depending upon the compiler used. If you get a syntax error or undefined reference with respect to a function name, check your compiler manual to see if the function is implemented and to check the syntax.

The various portions of the operating system have some standard features that apply to the function usage. First of all, the GEMDOS, BIOS, and XBIOS functions are all defined as macros in the header file OSBIND.H (see Appendix B). Therefore, any program that uses these functions must include OSBIND.H at the start of the program.

The VDI function parameters follow a specific format. All parameter lists for VDI functions start with a handle parameter. This handle refers to the workstation on which the function has an effect. The only functions that do not follow this format are **v_opnwk()** and **v_opnvwk()**, as they create workstations.

All VDI function names begin with "v_" characters. Whenever a point is required by a VDI function, it is passed in an array. The x coordinate is always followed by the y coordinate. For example, in an array called **sample**, element **sample[0]** would be the first x coordinate and **sample[1]** its corresponding y coordinate. Subsequent points are placed sequentially in the array. Whenever a rectangle is required, it is passed as a pair of points in an array (i.e., four elements). The first point defines the location of the upper left corner of the rectangle, and the second point provides the coordinates for the lower right corner.

The last comment with respect for the VDI refers to the input modes. The VDI allows two input modes for the input devices: request and sample. A request function waits until input is available from the device. A sample function tests the device to see if data is available. If it is not, the program continues and the function indicates that no data was found. If data is available, it is acted upon and this status condition is returned by the function.

The AES uses a different method of specifying a rectangular area than the VDI. The AES uses four values providing the coordinates of the upper left corner of the rectangle and its width and height. Unlike VDI coordinates, AES coordinates always refer to screen pixels and are device-dependent. Therefore, the width and height of a rectangle extend to the right and down from the coordinates given. A rectangular area required by an AES function will be supplied through four individual parameters.

The functions are listed alphabetically below. In cases where the function name includes the underscore character (_), the name is alphabetized as if the underscore did not exist. Following the function descriptions is a list of the function grouped by usage.

WORD **appl_exit ( )**

Remove an application from the AES. When this function is called, the calling application is removed from GEM AES and the AES cleans up the environment variables and arrays. A positive integer is returned if successful; otherwise 0 is returned.

```
WORD appl_find (ap_fpname)
        char        *ap_fpname;
```

Find the application ID of another application in the system. Parameter **ap_fpname** points to a string of eight characters containing the filename of the application to locate. The string *must* contain eight characters. If the filename is less than eight characters, the programmer must fill the remaining characters with blank spaces. A −1 returned value means that the application could not be located by the AES.

WORD **appl_init( )**

Initialize the internal arrays used by the GEM AES. An application ID number is returned. If the ID is −1, the function call was unsuccessful.

```
WORD appl_read (ap_rid, ap_rlength, ap_rpbuff)
        WORD    ap_rid,
                ap_rlength;
        char    *ap_rpbuff;
```

Read a message waiting for an application. When a message event occurs, the event function fills a 16-byte message buffer. If more than 16 bytes are in the message, this function is used to read the remaining data. Parameter **ap_rid** is the ID number of the application that sent the message, and **ap_rlength** is the number of bytes to read. The **ap_rpbuff** string is a buffer to hold the data being read. A returned value of 0 indicates an error.

```
WORD appl_tplay (ap_tpmen, ap_tpnum, ap_tpscale)
        char    *ap_tpmem;
        WORD    ap_tpnum,
                ap_tpscale;
```

Play a portion of pre-recorded GEM AES user actions. After a user's actions have been recorded, this function can be used to replay them. The parameter **ap_tpmem** points to the user events to be played. Parameter **ap_tpnum** indicates the number of actions to be played. Parameter **ap_tpscale** sets the replay speed ranging in value from 1 to 10,000. A value of 50 is half speed, 100 is full speed, and 200 is twice speed. The returned value always equals 1.

```
WORD appl_trecord (ap_trmen, ap_trcount)
        char    *ap_trmem;
        WORD    ap_trcount;
```

Record a specified number of user actions. As the user interacts with the application, this function records the user's actions. Parameter **ap_trmem** points to a segment of memory where the actions will be stored, and **ap_trcount** is the number of actions to record. The function returns the number of actions actually recorded.

Each action is stored as a two-part data item. The first part consists of two bytes (16 bits) to indicate the event code as follows:

0	timer event
1	button event
2	mouse event
3	keyboard event

The next four bytes contain information regarding the event in the following format:

timer event	the number of milliseconds elapsed
button event	low word: button state as defined in the text high word: number of clicks
mouse event	low word: mouse x coordinate high word: mouse y coordinate
keyboard event	low word: keyboard character code high word: keyboard state (see text)

WORD **appl_write (ap_wid, ap_wlength, ap_wpbuff)**
    WORD **ap_wid,**
    ap_length;
    char ***ap_wpbuff;**

Write a message to the AES. This function writes **a** message to an application in the system. The message is in the buffer **ap_wpbuff**. The number of bytes is specified by **ap_wlength** and the application ID to write to is **ap_wid**. A returned value of 0 indicates an error.

long **Bconin(dev)**
    WORD **dev;**

Wait for a character to be returned from the device specified by **dev**. The character is returned in the low WORD of the long value. If bit 3 of the system variable **conterm** is set, the high WORD contains the value of the system variable **kbshift** when the key was pressed (see Appendix D, System Addresses).

VOID **Bconout (dev, c)**
    WORD **dev, c;**

Wait until the character **c** has been written to device **dev**.

WORD **Bconstat (dev)**
    WORD **dev;**

Get the input status of a character device. The function returns −1 if characters are available, or 0 if not. The device numbers are:

Number	*Device*
0	PRT: — parallel printer port
1	AUX: — auxiliary RS-232 port
2	CON: — console/keyboard
3	MIDI port
4	Keyboard port (Atari extension)
5	Raw console output

For these devices, the legal BIOS operations are:

Operation	*PRT:*	*AUX:*	*CON:*	*MIDI*	*Kbd*	*Raw*
Bconstat( )	no	yes	yes	yes	no	no
Bconin( )	yes	yes	yes	yes	no	no
Bconout( )	yes	yes	yes	yes	yes	yes
Bcostat( )	yes	yes	yes	yes	yes	no

The MIDI device has an interrupt-driven input buffer of 80 characters. The keyboard device (4) is output only and can be used to configure the intelligent keyboard (see Atari documentation for further details). The raw console device outputs characters to the screen without interpretation of control or escape sequences.

long Bcostat (dev)
    WORD **dev;**

Return −1 if device **dev** is ready for output, or 0 if not. See function **Bconstat( )** for values of **dev**.

VOID **Bioskeys ( )**

Restore the initial system settings for the keyboard translation tables (see the **Keytbl( )** function).

WORD **Cauxin ( )**

Read a character from the auxiliary port (handle 1). This port is normally the serial port.

WORD **Cauxis ( )**

Check the status of the AUX: input. If a character is ready to be read, 0xFFFF is returned. Otherwise 0 is returned.

WORD **Cauxos ( )**

Check the status of the AUX: output. If it is ready to receive a character, 0xFFFF is returned. Otherwise 0 is returned.

VOID **Cauxout (c)**
    WORD **c;**

Write the character **c** to AUX:, the auxiliary port (handle 1). The high eight bits are reserved and must be 0. This port is usually the serial port.

long Cconin( )

Read and echo a character from the standard input. If the standard input device is the console, the long value returned contains both the ASCII and the console scan codes in the following format:

Bits	*Contents*
0–7	ASCII character code
8–15	All 0
16–23	Scan code or 0 if not console
24	Right shift key
25	Left shift key
26	Control key
27	Alternate key
28	Caps Lock key
29	Right mouse button or Clr/Home key
30	Left mouse button or Insert key
31	Reserved

For bits 24–31, if the bit is set to 1, the key is depressed.

WORD **Cconis ( )**

Check the status of the standard input. If a character is available, the function returns 0xFFFF. Otherwise 0 is returned.

WORD **Cconos ( )**

Check the status of the standard output. If the standard output is ready to receive a character, this function returns 0xFFFF. Otherwise 0 is returned.

```
VOID Cconout (c)
     WORD     c;
```

Write the character with ASCII value in **c** to the standard output. The high eight bits are reserved and must be 0.

```
VOID Cconrs (buf)
     char     *buf;
```

Read a string from the standard input. Common line editing characters are interpreted:

Character	*Function*
Return or Ctrl-J	End of line
Ctrl-H or Delete	Remove last character
Ctrl-U or Ctrl-X	Erase entire line
Ctrl-R	Retype the line
Ctrl-C	Terminate the process

The element **buf[0]** contains the number of characters typed. Element **buf[1]** contains the number of character read on exit. Element **buf[2]** is the first character of the string entered. The string may or may not be null-terminated.

VOID **Cconws (str)**
    char *str;

Write the null-terminated string **str** to the standard output.

long Cnecin ( )

Read a character from the standard input. If the input device is CON:, no echoing is done and control characters are interpreted.

WORD **Cprnos ( )**

Check the status of the PRN: output. If it is ready to receive a character, 0xFFFF is returned. Otherwise 0 is returned.

VOID **Cprnout (c)**
    WORD **c;**

Write the character **c** to **PRN;** the printer port (handle 2). The high eight bits are reserved and must be 0.

long Crawcin ( )

Read a character from the standard input (CON:, handle 0) without echo.

long Crawio (w)
    WORD **w;**

Read or write to the standard input/output. If **w** is 0x00FF, a character is read. If no character is available, 0 is returned. If **w** is not 0x00FF, the character **w** is written to the standard output.

WORD **Cursconf (op, data)**
    WORD **op, data;**

Configure the VT52 emulator cursor. The configuration function to perform depends on the value of **op** as follows:

op	*Function*
0	Hide cursor
1	Show cursor
2	Set blinking cursor
3	Set nonblinking cursor
4	Set blink rate according to value in data
5	Return current cursor blink rate

The cursor blink rate is based on the vertical blank interrupt rate. On the monochrome monitor, this occurs at 70 Hz. On the color monitor, this occurs at 60 Hz in NTSC mode or 50 Hz in PAL mode. The time for the cursor to turn off and back on again is twice the value of **data** divided by the vertical blank rate.

WORD **Dcreate (pathname)**
    char ***pathname;**

Create a directory specified by the string pointed to by **pathname.** The string gives the complete path for the new directory. 0 is returned if successful, otherwise an error value is returned.

WORD **Ddelete (pathname)**
    char ***pathname;**

Delete the directory specified by the path name pointed to by **pathname.** The directory must be empty and may not be the the "." or ".." special directories. 0 value is returned on success, or an error value is returned.

VOID **Dfree (buf, driveno)**
    struct disk_info ***buf;**
    WORD **driveno;**

Get the allocation information about the drive specified in **driveno.** Drive 0 is the default drive, 1 is drive A, 2 is drive B, etc. The **disk_info_buf** structure is filled by the function and is defined:

```
struct disk_info {
    long b_free;      /* # of free clusters on drive */
    long b_total;     /* total # of clusters on drive */
    long b_secsiz;    /* # of bytes in a sector */
    long b_clsiz;     /* # of sectors in a cluster */
}
```

With this information, the application can determine the amount of available space on the drive.

WORD **Dgetdrv ( )**

Returns the number of the current drive with 0 = A, 1 = B, etc.

```
VOID Dgetpath (buf, driveno)
    char      *buf;
    WORD      driveno;
```

Get the current directory path for a drive. The path is placed into **buf.** The drive number is 0 for the default drive, 1 for drive A, 2 for drive B, etc. The application must supply enough space in **buf** to hold the path name.

```
VOID Dosound (ptr)
    char      *ptr;
```

Initiate the sound daemon's "program counter" to the address contained in **ptr.** A daemon is an independent process. The series of bytes pointed to by **ptr** contain instructions and data for the sound daemon. The instructions are:

Instruction	*Function*
0x00–0x0F	Put the next byte into the sound register. 0x00 puts the data into register 0, 0x01 puts it into register 1, etc.
0x80	Put the next byte into the temporary register (temp_reg)
0x81	Uses the next three bytes, calling them reg_no, byte1, and byte2, where reg_no is the register to use, byte1 is the increment value, and byte2 is the termination value. The following procedure is performed: Until value in temp_reg = byte2 Put value from temp_reg into reg_no Add byte1 to temp_reg Repeat

Instruction	*Function*
0x82-0xFE	The next byte contains the number of 1/50 second time units to wait before continuing
0xFF	If the next byte is 0, the second daemon stops. Otherwise the value is used like instructions 0x82 to 0xFE

long Drvmap( )

Return a value with bit settings indicating which drives are available. If the bit is set to 1, the drive is available at that location. Bit 0 corresponds to drive A, bit 1 for drive B, etc.

long Dsetdrv (drv)
WORD **drv;**

Set the default drive. The value in **drv** determines the new default drive with 0 = A, 1 = B, . . . , 15 = P. The value returned has its low 16 bits set to 1 for each drive available (bit 0 = A, bit 1 = B, etc.). A drive is available if its directory has been used. Only 16 drives are available.

WORD **Dsetpath (path)**
char ***path;**

Set the current directory path to the path name pointed to by **path.** 0 is returned if successful; otherwise an error value is returned.

WORD **evnt_button (ev_bclicks, ev_bmask, ev_bstate, ev_bmx, ev_bmy, ev_bbutton, ev_bkstate)**
WORD **ev_bclicks,**
ev_bmask,
ev_bstate,
***ev_bmx, *ev_bmy,**
***ev_bbutton,**
***ev_bkstate;**

Wait until a mouse button is clicked. Depending upon the input parameters, this function responds only to a certain combination of mouse buttons and a certain number of clicks. The returned value is the number of times the specified state had been achieved. The parmeter are used as follows:

ev_bclicks	the number of clicks needed for the event
ev_bmask	the mask for the buttons of interest. GEM AES can handle up to 16 buttons where bit 0 refers to the furthest left button. When the bit is set to 1, that button is used.

ev_bstate	the state to detect for each button. The bits are in the same order as ev_bmask and have the following meaning: 0 = button up 1 = button down
ev_bmx	the x coordinate of the mouse pointer when the event occurred
ev_bmy	the y coordinate of the mouse pointer when the event occurred
ev_button	the state of the mouse buttons upon exit from the routine. The bit settings are the same as ev_bstate.
ev_bkstate	the keyboard state when the function returned. The bits in this value have the following meaning: Bit 1 right shift Bit 2 left shift Bit 3 Control key Bit 4 Alternate key If the bit is set, the key has been pressed.

The returned value of the function is the number of times the button(s) actually entered the desired state. This value is never less than 1 or greater than **ev_bclicks.**

WORD **evnt_dclick (ev_dnew, ev_dgetset)**
    WORD **ev_dnew,**
    ev_dgetset;

Set or read the double-click speed for the mouse button. When **ev_dgetset** is 1, the double-click speed is set to the value in **ev_dnew** and this value is returned by the function. Otherwise, the current double_click speed is returned by the function. The speed can range from 0 as slowest to 4 as fastest.

WORD **evnt_keybd ( )**

Wait for a keyboard event. The application processing stops until a keyboard event for the application is reported by the AES. The returned value is the keyboard code (see Appendix C) for the character typed.

WORD **evnt_mesag (ev_mgpbuff)**
    char ***ev_mgpbuff;**

Wait for a message event. When a message is sent to an application, the AES issues a message event to the receiving application. This function acknowledges the message event and reads a standard 16-byte message from the AES. The 16 bytes are placed in the buffer

pointed to by **ev_mgpbuff**. The formats for the pre-defined AES messages are given in Appendix E.

```
WORD evnt_mouse (ev_moflags,
                 ev_mox, ev_moy, ev_mowidth, ev_moheight,
                 ev_momx, ev_momy, ev_mobutton, mv_mokstate)
        WORD     ev_moflags,
                 ev_mox, ev_moy,
                 ev_mowidth, ev_moheight,
                 *ev_momx, *ev_momy,
                 *ev_mobutton,
                 *ev_mokstate;
```

Wait for the mouse to enter or leave a specified rectangle. The parameters are as follows:

ev_moflags	a value of 0 means to wait for the mouse to enter the rectangle. A value of 1 means to wait for the mouse to exit.
ev_mox	the x coordinate of the upper left corner of the rectangle in screen coordinates.
ev_moy	the y coordinate of the upper left corner
ev_mowidth	the width of the rectangle in pixels
ev_moheight	the height of the rectangle in pixels
ev_momx	the x coordinate of the point where the mouse event occurred
ev_momy	the y coordinate of the point where the mouse event occurred
ev_mobutton	the state of the mouse buttons when the event occurred. Bit 0 corresponds to the furthest left button. A 1 value means the button was pressed.
ev_mokstate	the status of the keyboard special function keys. If a bit is 1, the key was pressed. The bits are used as follows: Bit 0 right shift Bit 1 left shift Bit 2 Control key Bit 3 Alternate key

The function always returns the value 1.

```
WORD evnt_multi (ev_mflags, ev_mbclicks, ev_mbmask,
                 ev_mbstate, ev_mm1flags, ev_mm1x, ev_mm1y,
                 ev_mm1width, ev_mm1height, ev_mm2flags,
                 ev_mm2x, ev_mm2y, ev_mm2width, ev_mm2height,
                 ev_mmgpbuff, ev_mtlocount, ev_mthicount, ev_mmox,
                 ev_mmoy, ev_mmobutton, ev_mmokstate,
                 ev_mkreturn, ev_mbreturn)
```

```
WORD        ev_mmflags,
            ev_mbclicks, ev_mbmask, ev_mbstate,
            ev_mm1flags, ev_mm1x, ev_mm1y,
            ev_mm1width, ev_mm1height,
            ev_mm2flags, ev_mm2x, ev_mm2y,
            ev_mm2width, ev_mm2height,
            ev_mtlocount, ev_mthicount,
            *ev_mmox, *ev_mmoy,
            *ev_mmobutton, *ev_mmokstate,
            *ev_mkreturn, *ev_mbreturn;
char        *ev_mmgpbuff;
```

Wait for any one of six possible events. This function combines **evnt_button( ), evnt_keybd( ), evnt_mesag( ), evnt_mouse( ), and evnt_timer( )** into one function call. **The event(s)** to wait for are specified by the bits in **ev_mflags.** Any combination of events is valid. The parameters are basically the same as those used for the various independent event functions, and are used as follows:

ev_mmflags — a set of bits indicating the event(s) to accept. If the bit is set to 1, the function responds to that event. The bits are specified as follows and have the associated defined constants:

Bit	*Constant*	*Value*	*Event*
0	MU_KEYBD	0x0001	keyboard
1	MU_BUTTON	0x0002	mouse button
2	MU_M1	0x0004	first mouse rectangle
3	MU_M2	0x0008	second mouse rectangle
4	MU_MESAG	0x0010	message
5	MU_TIMER	0x0020	timer

ev_mbclicks — the number of clicks for a button event

ev_mbmask — the mouse buttons of interest. Bit 0 corresponds to the furthest left button. A bit set to 1 means that button was used.

ev_mbstate — the state of the mouse buttons. A bit set to 1 means the button is down.

ev_mm1flags — first rectangle event flag. A value of 0 causes an event when the mouse enters the rectangle

ev_mm1x, ev_mm1y, ev_mm1width, and ev_mm1height — these parameters define the location and size of the first mouse event rectangle

ev_mm2flags — second rectangle event flag

ev_mm2x, ev_mm2y, ev_mm2width, and ev_mm2height — these parameters define the location and size of the second mouse event rectangle

ev_mgpbuff — a pointer to a 16-byte buffer to hold the message in the case of a message event

ev_mtlocount and ev_mthicount — the low and high words of a long integer used to set the number of milliseconds to wait for a timer event

ev_mmox and ev_mmoy — the x and y coordinates of the mouse pointer when the mouse event occurred

ev_mmobutton — the state of the mouse buttons when the event occurred

ev_mokstate — the state of the keyboard when the event occurred. The bits have the following meaning:

Bit 0	right shift
Bit 1	left shift
Bit 2	Control key
Bit 3	Alternate key

ev_mkreturn — keycode for the key pressed in a keyboard event

ev_mbreturn — the number of times the mouse buttons entered the desired state

The function returns a value with the bit set to indicate the event that had occurred. The bit representations are the same as for parameter **ev_mflags.**

```
WORD evnt_timer (ev_tlocount, ev_thicount)
        unsigned WORD ev_tlocount,
                      ev_thicount;
```

Wait for a specified amount of time to pass. The amount of time is given as a long integer value to indicate length of the time interval in milliseconds. The long integer (four bytes) is divided into the unsigned WORD (i.e., int) values of two bytes each. The low word is passed in parameter **ev_tlocount** and the high word is in **ev_thicount.** The function always returns a value of 1.

```
WORD Fattrib (fname, wflag, attribs)
        char    *fname;
        WORD    wflag;
        WORD    attribs;
```

Get or set the file attribute bits. The file is specified by the name given by **fname.** If **wflag** is 0, the file's current attributes are returned. If

wflag is 1, the file's attributes are set to bit settings of **attribs** as follows:

Bit	*Attribute*
0	Read only
1	Hidden
2	System
3	11 byte volume label
4	Subdirectory
5	File has been written to and closed

```
WORD Fclose (fhandle)
        WORD    fhandle;
```

Close the file associated with **fhandle.** 0 is returned if successful; otherwise an error value is returned.

```
WORD Fcreate (fname, attribs)
        char    *fname;
        WORD    attribs;
```

Create a file with the filename **fname** and the attributes given by **attribs.** The bits of **attribs** have the following meaning:

Bit	*Description*
0	File set to read only
1	File hidden from directory search
2	File set to SYSTEM
3	File contains 11 byte volume label

If the bit is set to 1, the file has that attribute.

The function returns a write only nonstandard handle to the file, which is a positive number. If an error occurred, a negative error value is returned.

```
VOID Fdatime (fhandle, timeptr, wflag)
        WORD    fhandle;
        long    *timeptr;
        WORD    wflag;
```

Get or set the date and time stamp for a file. The file is referred by its handle given in **fhandle.** If the parameter **wflag** is 0, the time stamp is

read into **timeptr.** If **wflag** is 1, the time stamp is set from **timeptr.** The high WORD of the value at **timeptr** is the time and the low WORD is the date. See the **Tgetdate( )** and **Tgettime( )** functions for the format of the date and time WORDs.

```
WORD Fdelete (fname)
        char      *fname;
```

Delete the file named by **fname.** A returned value of 0 indicates success; otherwise a negative error number is returned.

```
WORD Fdup (fhandle)
        WORD   fhandle;
```

Duplicates a standard handle. A standard handle has a value from 0 through 5. The handle to duplicate is given by parameter **fhandle.** The function returns a nonstandard handle or a negative error number.

```
WORD Fforce (stdh, nonstdh)
        WORD   stdh,
               nonstdh;
```

Force the standard handle **stdh** to point to the same file or device as the nonstandard handle **nonstdh.** The returned value is 0 if successful; otherwise an error number is returned.

```
long Fgetdta ( )
```

Returns the address of the current Disk Transfer Area (DTA) used by function **Fsfirst( )** and **Fsnext( ).**

```
WORD Flopfmt (buf, filler, drvno, spt, trackno, sideno,
          interlv, magic, virgin)
        WORD     *buf;
        long     filler;
        WORD     drvno, spt, trackno, sideno,
                 interlv, virgin;
        long     magic;
```

Format one track on a floppy disk. The buffer pointed to by **buf** must be large enough to hold an entire track image (8K for nine sectors per track). Parameters **drvno, spt, trackno,** and **sideno** give the drive number (0 or 1), the number of sectors per track (usually nine), the track number (0 through 79), and the side number (0 or 1), respectively. The parameter filler is unused. The sector interleave factor (usually 1) is

given as **interlv.** The value for magic is the special number 0x87654321. The data in the sectors is filled with the value in virgin. The high bit of virgin must not be set, and the value 0 should be avoided. The recommended virgin value is 0xE5E5.

The function returns 0 if successful, or a negative error value if not. A null-terminated list of bad sector numbers (one WORD each) is returned in the buffer. The list is not necessarily in numerical order. If no bad sectors were found, the first WORD will be 0.

```
WORD Floprd (buf, filler, drvno, sectno, trackno, sideno, count)
        WORD    *buf;
        long    filler;
        WORD    drvno, sectno, trackno, sideno, count;
```

Read sectors from a floppy disk. The function reads count number of sectors into the word-aligned buffer pointed to by **buf.** Parameters **drvno, sectno, trackno,** and **sideno** give the drive number (0 or 1), the sector number (1 through 9), the track number (0 through 79), and the side number (0 or 1), respectively. The parameter filler is unused. The function returns 0 if successful, or a negative error value if not.

```
WORD Flopver (buf, filler, drvno, sectno, trackno, sideno, count)
        WORD    *buf;
        long    filler;
        WORD    drvno, sectno, trackno, sideno, count;
```

Verify a floppy disk. The function reads count number of sectors from the word-aligned buffer pointed to by **buf.** The buffer must be 1,024 bytes in length. Parameters **drvno, sectno, trackno,** and **sideno** give the drive number (0 or 1), the sector number (1 through 9), the track number (0 through 79), and the side number (0 or 1), respectively. The parameter filler is unused.

The function returns 0 if successful, or an negative error value if not. A null-terminated list of bad sector numbers (one WORD each) is returned in the buffer. The list is not necessarily in numerical order. If no bad sectors were found, the first WORD will be 0.

```
WORD Flopwr (buf, filler, drvno, sectno, trackno, sideno, count)
        WORD    *buf;
        long    filler;
        WORD    drvno, sectno, trackno, sideno, count;
```

Write sectors to a floppy disk. The function writes **count** number of sectors from the word-aligned buffer pointed to by **buf.** Parameters **drvno, sectno. trackno,** and **sideno** give the drive number (0 or 1), the

sector number (1 through 9), the track number (0 through 79), and the side number (0 or 1), respectively. The parameter filler is unused. The function returns 0 if successful, or an negative error value if not.

```
WORD Fopen (fname, mode)
        char        *fname;
        WORD        mode;
```

Open the file named **fname** with the access set to mode. The access may be 0 for read only, 1 for write only, or 2 for read or write. The function returns a positive nonstandard handle to the file. A negative returned error value is returned on an error.

```
WORD form_alert (fo_adefbttn, fo_astring)
        WORD        fo_adefbttn;
        char        *fo_astring;
```

Display an alert box. This function handles the drawing and interaction required to initialize the screen, draw the alert box, allow user interaction, restore the screen, and report the exit button selected to the application. Parameter **fo_adefbttn** specifies the default exit button as follows:

0	No default exit button
1	First exit button
2	Second exit button
3	Third exit button

Parameter **fo_astring** points to a string defining the format for the alert box. The string consists of three sections. Each section is surrounded by a set of square brackets ([ ]). The first section specifies the icon to use where:

0	No icon,
1	The NOTE icon,
2	The WAIT icon, and
3	The STOP icon.

The second portion of the string contains the message text. The text may consist of up to four lines of 32 characters per line. To signal the end of a line in the message text, use the vertical bar (|) character.

The last portion of the string contains the text for up to three exit buttons with 20 characters each. The vertical bar (|) is again used to

separate the text for each button. The format for the **fo_astring** parameter is given below:

"[(icon #)][(message test)][(exit buttons)]"

The function returns the number of the button used to exit the alert box.

```
WORD form_center (fo_ctree, fo_cx, fo_cy, fo_cw, fo_ch)
        OBJECT    *fo_ctree;
        WORD      *fo_cx, *fo_cy, *fo_cw, *fo_ch;
```

Determine the coordinates to center a dialog box on the screen. The object tree for the dialog box is pointed to by **fo_ctree.** Upon return, **fo_cx, fo_cy, fo_cw,** and **fo_ch** define the position and size of a rectangle, centered on the screen, for the dialog box. The function always returns the value 1.

```
WORD form_dial (fo_diflag,
            fo_dilittlx, fo_dilittly, fodilittlw, fo_dilittlh,
            fo_dibigx, fo_dibixy, fo_dibigw, fo_dibigh)
        WORD      fo_diflag,
                  fo_dilittlx, fo_dilittly, fo_dilittlw, fo_dilittlh,
                  fo_dibigx, fo_dibigy, fo_dibigw, fo_dibigh;
```

Dialog box display function. This function performs four distinct operations associated with displaying a dialog box. First, the function can reserve space for the dialog box. Second, the function can draw an expanding box. Third, the function can draw a shrinking box. And finally, the function can release the space reserved for the dialog box. Only the first and last of these operations must be done before displaying the dialog box.

The smallest size for the dialog box is defined by **fo_dilittlx, fo_dilittly, fo_dilittlw,** and **fo_dilittlh,** which give the x and y coordinates, width, and height, respectively. The largest size (and the size used to reserve screen space) is given by **fo_dibigx, fo_dibigy, fo_dibigw,** and **fo_dibigh** for the x and y coordinates, width, and height, respectively. All coordinates are relative to the screen.

The actual operation performed is specified by the value for **fo_diflag** using the defined constants:

Constant	*Value*	*Function*
FMD_START	0	Reserve screen space for dialog box
FMD_GROW	1	Draw expanding box
FMD_SHRINK	2	Draw shrinking box
FMD_FINISH	3	Release reserved space and cause application to redraw screen

A returned value of 0 indicates an error.

```
WORD form_do (fo_dotree, fo_dostartob)
        OBJECT   *fo_dotree;
        WORD     fo_dostartob;
```

Cause the Form Manager to monitor the user's interactions with a form. Parameter **fo_dotree** points to the object tree containing the form. Paramenter **fo_dostartob** is the object index of the starting object which will be active when the form appears. This object must be an editable text field. A value of −1 indicates that the form does not contain an editable text field. When **form_do( )** is called, control is passed to the AES until the user creates an exit condition (see text). The function returns the object index of the object that caused the exit condition.

```
WORD form_error (fo_enum)
        WORD     fo_enum;
```

Display an error box. This is a pre-defined dialog box that displays a message based upon the error number specified in **fo_enum.** GEMDOS has predefined numbers representing system errors. However, these values are usually returned by the DOS functions as negative values so the application must use the absolute value of the returned error value.

The value returned by this function is the number of the exit button selected. On the Atari, only one exit button is displayed in the error box.

```
long Fread (fhandle, count, buf)
    WORD     fhandle;
    long     count;
    char     *buf;
```

Read from the file indicated by **fhandle.** The number of bytes to read is given by **count,** and the data is placed in the buffer pointed to by **buf.** The actual number of bytes read is the return value. A value of 0 indicates the end of file, and a negative value is an error value.

```
WORD Frename (zero, oldname, newname)
        WORD     zero;
        char     *oldname,
                 *newname;
```

Rename a file from **oldname** to **newname.** The new name of the file must not exist on the disk. The new file may be in another directory.

The parameter **zero** is always 0. A value of 0 is returned if successful; otherwise an error number is returned.

```
long Fseek (offset, fhandle, seekmode)
    long    offset;
    WORD    fhandle;
    WORD    seekmode;
```

Set the current file pointer position with the file specified by **fhandle.** The **offset** is the number of bytes to move. The **seekmode** is the position from which to count the **offset.** The **seekmode** has the following values:

Seekmode	*Offset Start Position*
0	From the beginning of the file
1	From the current position
2	From the end of the file

The new, absolute pointer position is returned.

```
WORD fsel_input (fs_iinpath, fs_iinsel, fs_iexbutton)
    char    *fs_iinpath,
            *fs_iinsel;
    WORD    *fs_iexbutton;
```

Display the file selector dialog box and allow the user to select a file. This function handles all procedures and data required to display and manage the file selector dialog box. The application supplies the default search path in string **fs_iinpath,** and the default filename in string **fs_iinsel.** Upon return, **fs_iinpath** contains the search path set by the user, and **fs_iinsel** contains the filename selected by the user. Parameter **fs_iexbutton** is set to the exit button used, where 0 is the Cancel button and 1 is the OK button. A value of 0 indicates an error has occurred.

```
VOID Fsetdta (addr)
    char    *addr;
```

Set the Disk Transfer Area (DTA) to the memory pointed to by **addr.** The DTA is used only by function **Fsfirst( )** and **Fsnext( )).**

```
WORD Fsfirst (fspec, attribs)
    char    *fspec;
    WORD    attribs;
```

Search for the first occurrence of the file given by **fspec.** The file specification may contain wildcard characters (? and *) in the filename, but not in the pathname. The bit settings of **attribs** determine which files are located. If **attribs** is 0, only "normal" files are found. The bit settings are:

Bit	*Attribute*
0	Read only
1	Hidden
2	System
3	11-byte volume label
4	Subdirectory
5	File has been written to and closed

When the file is located, the function fills the Disk Transfer Area (DTA) with information about the file. The DTA is a 44-byte data structure with the following format:

Byte	*Length*	*Contents*
0	20 bytes	Reserved for OS
21	1 byte	File attributes (as above)
22	2 bytes	File time stamp
24	2 bytes	File date stamp
26	4 bytes	File size
30	14 bytes	File name and extension

The function returns 0 if a file was found; otherwise a negative error number is returned.

WORD **Fsnext** ( )

Search for the next occurrence of a file. The search parameters are initially set by **Fsfirst( ).** If a file is found, the DTA is filled as in **Fsfirst( ).** The function returns 0 if a file is found; otherwise an error number is returned.

long Fwrite (fhandle, count, buf)
```
    WORD    fhandle;
    long    count;
    char    *buf;
```

Write the data from the memory location pointed to by **buf** to the file specified by **fhandle.** The number of bytes to write is given by **count.** A

positive value returned gives the number of bytes actually written. A negative value is an error number.

BPB ***Getbpb (drv)**
drv;

Get the BIOS Parameter Block (BPB) of the drive given by **drv.** The drive numbers are: 0 is A, 1 is B, etc. The function returns a pointer to the BPB, or NULL if the BPB could not be determined. The BPB is defined:

```
typedef strubt bios_pb {
    WORD recsiz,    /* physical sector size in bytes (512) */
         clsiz,     /* cluster size in sectors (2) */
         clsizb,    /* cluster size in bytes (1024) */
         rdlen,     /* root directory length in sectors */
         fsize,     /* FAT size in sectors */
         fatrec,    /* sector # of 1st sector of 2nd FAT */
         datrec,    /* sector # of 1st data sector */
         numcl,     /* # of data clusters on disk */
         bflags;    /* flags */
} BPB;
```

The usage of the BPB and its contents extend beyond the scope of this book. Refer to the GEMDOS documentation for more details.

VOID **Getmpb (p_mpb)**
struct mpb ***p_mpb;**

Get a Memory Parameter Block (MPB). See Appendix D, System Addresses, for information on the **mpb** structure.

WORD **Getrez( )**

Return the screen's current resolution as:

0	Low resolution (320 × 200)
1	Medium resolution (640 × 200)
2	High resolution (640 × 400)

long Gettime( )

Return the current date and time. The bits in the long value returned

have the following interpretation:

Bits	*Meaning*
0–4	Seconds divided by 2 (0–29)
5–10	Minutes (0–59)
11–15	Hours (0–23)
16–20	Day in month (1–31)
21–24	Month (1–12)
25–31	Years since 1980 (0–119)

```
char Giaccess (data, regno)
    char    data;
    WORD    regno;
```

Access the registers on the AY-3-8910 sound chip. The register to access is given by **regno.** If **regno** has its high bit set (logically ORed with 0x80), the function writes the value in data into the register. Otherwise the function returns the current value in the register. See Chapter 6 for details regarding sound generation.

The sound chip is used by the operating system for other functions. Therefore a program accessing the sound chip must perform its changes uninterrupted. This function provides that facility and should be used.

```
WORD graf_dragbox (gr_dwidth, gr_dheight,
            gr_dstartx, gr_dstarty,
            gr_dboundx, gr_dboundy, gr_dboundw, gr_dboundh,
            gr_dfinishx, gr_dfinishy)
        WORD    gr_dwidth, gr_dheight,
                gr_dstartx, gr_dstarty,
                gr_dbounx, gr_dboundy,
                gr_dboundw, gr_dboundh,
                *gr_dfinishx, *gr_dfinishy;
```

Allow the user to drag a box outline on the screen. When this function is called, the AES places a rectangle outline on the screen with the width **gr_dwidth** and height **gr_dheight.** The upper left corner of the outline starts at coordinates **gr_dstartx** and **gr_dstarty.** The application can limit the movement of the drag box by specifying the boundary rectangle in **gr_dboundx, gr_dboundy, gr_dboundw,** and **gr_dboundh.** When the user releases the mouse button, the function puts the coordinates of the upper left corner into parameters **gr_dfinishx** and **gr_dfinishy.** A returned value of 0 indicates an error.

```
WORD graf_growbox (gr_gstx, gr_gsty, gr_gstwidth, gr_gstheight,
        gr_gfinx, gr_gfiny, gr_gfinwidth, gr_gfinheight)
    WORD    gr_gstx, gr_gsty, gr_gstwidth, gr_gstheight,
            gr_gfinx, gr_gfiny, gr_gfinwidth, gr_gfinheight;
```

Draw an expanding box outline. This routine places a box outline on the screen at the location and size given by **gr_gstx**, **gr_sty**, **gr_gstwidth**, and **gr_gstheight**. The box then expands to its final size and location given by **gr_gfinx**, **gr_gfiny**, **gr_gfinwidth**, and **gr_gfinheight**. A returned value of 0 indicates an error.

```
WORD graf_handle (gr_hwchar, gr_hhchar, gr_hwbox, gr_hhbox)
    WORD    *gr_hwchar, *gr_hhchar,
            *gr_hwbox, *gr_hhbox;
```

Request the handle of the currently open physical workstation for the screen being used by the VDI. The parameters to the function are filled with the size of the system font characters as follows:

gr_hwchar	the width of the character cell
gr_hhchar	the height of the character cell
gr_hwbox	the width of a box large enough to hold a system font character
gr_hhbox	the height of a box large enough to hold a system font character

All of the above values are measured in screen pixels.

```
WORD graf_mbox (gr_mwidth, gr_mheight, gr_msourcex,
        gr_msourcey, gr_mdestx, gr_mdesty)
    WORD    gr_mwidth, gr_mheight,
            gr_msourcex, gr_msourcey,
            gr_mdestx, gr_mdesty;
```

Display a box outline moving from one location to another on the screen. The box has a set size given by **gr_mwidth** and **gr_mheight** which does not change. The starting location (of the upper left corner) is given by **gr_msourcex** and **gr_msourcey**. The final location is given by **gr_mdestx** and **gr_mdesty**. A returned value of 0 indicates an error.

```
WORD graf_mkstate (gr_mkmx, gr_mkmy, gr_mkmstate,
        gr_mkkstate)
    WORD    *gr_mkmx, *gr_mkmy,
            *gr_mkmstate, *gr_mkkstate;
```

Obtain the current mouse location, mouse button state, and keyboard state. The current mouse location is returned through parameters **gr_mkmx** and **gr_mkmy**. The button state is returned through parameter **gr_mkmstate**. The bits of this value indicate the position of the button and its state. If the bit is set to 1, the button is currently down. Bit 0 corresponds to the furthest left button, bit 1 is the second button from the left, and so on.

The keyboard state is returned through parameter **gr_mkkstate**. The bit settings for this value are:

Bit	*Key*
0	Right shift
1	Left shift
2	Control
3	Alternate

If the bit is set to 1, the key is currently being pressed.

This function always returns the value 1.

```
WORD graf_mouse (gr_monumber, gr_mofaddr)
        WORD      gr_monumber;
        MFORM     *gr_mofaddr;
```

Change the mouse form. This function tells the AES which mouse form to use. The mouse form is given by the value in **gr_monumber** based upon the following constants:

Constant	*Value*	*Shape*
ARROW	0	Arrow
TEXT_CRSR	1	Text cursor
HOURGLASS	2	Bumble bee
POINT_HAND	3	Hand with pointing finger
FLAT_HAND	4	Open hand
THIN_CROSS	5	Thin cross hairs
THICK_CROSS	6	Thick cross hairs
OUTLN_CROSS	7	Outlined cross hairs
USER_DEF	255	Form defined by gr_mofaddr
M_OFF	256	Turn off mouse display
M_ON	257	Turn on mouse display

Calls to turn the mouse display on or off may be nested. Therefore each call to turn the mouse off must be balanced by a call to turn the mouse on before it will reappear, and vice versa.

If the mouse form selected is USER_DEF, the function refers to the mouse form definition block pointed to by **gr_mofaddr.** The structure of the mouse form definition block is:

```
typedef struct mfstr {
    WORD    mf_xhot;
    WORD    mf_yhot;
    WORD    mf_nplanes;
    WORD    mf_fg;
    WORD    mf_bg;
    WORD    mf_mask[16];
    WORD    mf_data[16];
} MFORM;
```

The fields **mf_xhot** and **mf_yhot** define the hot spot of the mouse form. The hot spot is the point the AES uses as the mouse location. The coordinates are measured from (0,0) being the upper left corner. The field **mf_nplanes** indicates the number of planes in the form. Fields **mf_fg** and **mf_bg** define the foreground and background colors to use when the mouse is drawn (see raster copy functions in the text). The **mf_mask** array is the raster for the mouse form mask and the **mf_data** array is the raster to the mouse form itself.

An application may use any mouse form required while the mouse is located within the top window's work area. However, once the mouse leaves this work area, it is the application's responsibility to restore the mouse form to the ARROW or HOURGLASS as appropriate. The application should use the **evnt_multi( )** function to detect when the mouse enters or leaves the window's work area.

A returned value of 0 indicates an error.

```
WORD graf_rubberbox (gr_rx, gr_ry, gr_rminwidth, gr_rminheight,
        gr_rlastwidth, gr_rlastheight)
    WORD    gr_rx, gr_ry,
            gr_rminsidth, gr_rminheight,
            *gr_rlastwidth, *gr_lastheight;
```

Maintain a "rubber box" on the screen. A rubber box is a rectangular outline that has its upper left corner at a fixed location (given by **gr_rx** and **gr_ry**), and the lower right corner dragged by the user. While the user keeps the mouse button depressed, the AES continues to track the mouse and draw the rubber box. When the user releases the mouse button, this function returns to calling application. The width

and height of the rubber box when the user releases the mouse button are returns through parameters **gr_rlastwidth** and **gr_rlastheight**, respectively. The minimum width and height that the box may have is passed through **gr_rminwidth** and **gr_rminheight**. A returned value of 0 indicates an error.

```
WORD graf_shrinkbox (gr_sfinx, gr_sfiny, gr_sfinwidth,
        gr_sfinheight, gr_sstx, gr_ssty, gr_sstwidth, gr_sstheight)
    WORD    gr_sfinx, gr_sfiny, gr_sfinwidth, gr_sfinheight,
            gr_sstx, gr_ssty, gr_sstwidth, gr_sstheight;
```

Draw a shrinking box outline. This routine places a box outline on the screen at the location and size given by **gr_sfinx**, **gr_sfiny**, **gr_sfinwidth**, and **gr_sfinheight**. The box then shrinks to its final size and location given by **gr_sstx**, **gr_ssty**, **gr_sstwidth**, and **gr_sstheight**. A returned value of 0 indicates an error.

```
WORD graf_slidebow (gr_slptree, gr_slparent, gr_slobject, gr_slvh)
    OBJECT  *gr_slptree;
    WORD    gr_slparent,
            gr_slobject,
            gr_slvh;
```

Track a sliding box within its parent box. When the user selects a slide box, the application calls this function to track the user's interactions. When the user releases the mouse button, the function reports the relative position of the slide box within its parent. The slide box and its slide bar must be objects in the tree pointed to by **gr_slptree**. The slide bar, with object index **gr_slparent**, must be the parent of the slide box, object index **gr_slobject**. The direction of tracking is set by **gr_slvh**:

0	Horizontal
1	Vertical

The value returned indicates the relative position of the slider within its parent box. This value will range from 0 to 1000. For horizontal sliders, 0 is the left edge, and for vertical sliders, 0 is the top edge.

```
WORD graf_watchbox (gr_wptree, gr_wobject, gr_winstate,
        gr_woutstate)
    OBJECT  *gr_wptree;
    WORD    gr_wobject,
            gr_winstate, gr_woutstate;
```

Track the mouse pointer in and out of a predefined box. The box is the rectangle associated with a particular object. The object tree is pointed to by **gr_wptree** and the object's index is given by **gr_wobject.** The state of the object while the mouse pointer is inside the box is specified by **gr_winstate.** A returned value of 0 indicates an error.

The state of the object when the mouse pointer is outside of the box is specified by **gr_woutstate.** The states are:

Bit	*Constant*	*Value*
None	NORMAL	0x0000
0	SELECTED	0x0001
1	CROSSED	0x0002
2	CHECKED	0x0004
3	DISABLED	0x0008
4	OUTLINED	0x0010
5	SHADOWED	0x0020

When the bit is set, the associated state is active. The states may be combined. The NORMAL state indicates no bits are set.

```
VOID Ikbdws (cnt, ptr)
        WORD        cnt;
        char        *ptr;
```

Write **cnt** bytes from the string at **ptr** to the intelligent keyboard processor. The **cnt** is the number of characters to write minus one.

```
VOID Initmous (type, paramp, vec)
                            type;
        struct param        *paramp;
        int                 (*vec) ( );
```

Initialize the mouse packet handler of the intelligent keyboard device. This function interfaces to the intelligent keyboard, which is beyond the scope of this book. See Atari documentation for further details. The function is presented here for reference. The parameter usage is:

Type	*Is the Type of Operation to be Performed as:*
0	Disable mouse
1	Enable mouse in relative mode
2	Enable mouse in absolute mode
3	Unused
4	Enable mouse in keycode mode

paramp	Points to a **param** structure (see below)
vec	Points to a mouse interrupt handler (see **kbdvbase()**).

The **param** structure contains:

```
struct param {
        char topmode;
        char buttons;
        char xparam;
        char yparam;
        WORD xmax, ymax;
        WORD xinitial, yinitial;
}
```

where:

topmode determines position of 0 y coordinate
0 = bottom of screen
1 = top of screen

button parameter for intelligent keyboard's "set mouse buttons" command
xparam and yparam have the following meanings depending on the mode:

Mode	*Meaning of xparam and xparam*
Relative	x and y interrupt threshold values
Absolute	x and y scale factors
Keycode	x and y delta factors

The remaining members of the param structure are used in absolute mode only, and give the maximum and initial x and y coordinates.

long Iorec (devno)
WORD **devno;**

Get the serial device input buffer descripter, which is:

```
struct iorec {
    char    *buf;       /* pointer to queue */
    WORD ibufsize,      /* size of queue in bytes */
         ibufhd,        /* head index of queue */
         ibuftl,        /* tail index of queue */
         ibuflow,       /* low data mark */
         ibufhigh;      /* high data mark */
}
```

The function returns a pointer to the input buffer descriptor immediately followed by the output buffer descriptor (of the same format).

The member **ibuftl** is the index of the last character to be placed in the queue, and **ibufhd** is the index of the last character to be removed from the queue. The queue is empty if **ibufhd** equals **ibuftl.** The ST requests that the sender stop transmitting when the number of characters in the queue equals **ibufhigh.** A request to the sender to resume transmitting is given when the number of characters falls below **ibuflow.** The output buffer is handled in a similar manner.

```
VOID Jdisint (intno)
     WORD    intro;
```

Disable interrupt number **intno** on the 68901. See function **Mfpint( )** for further details.

```
VOID Jenabint (intno)
     WORD    intno;
```

Enable interrupt number **intno** on the 68901. See function **Mfpint( )** for further details.

long Kbdvbase ( )

Get the list of the system vectors. The function returns a pointer to a structure of the following format:

```
struct kbdvecs {
     WORD (*midivec) ( ),      /* MIDI input */
          (*vkbderr) ( ),      /* keyboard error */
          (*vmiderr) ( ),      /* MIDI error */
          (*vstatvec) ( ),     /* ikbd status packet */
          (*mousevec) ( ),     /* mouse packet */
          (*clockvec) ( ),     /* clock packet */
          (*joyvec) ( ),       /* joystick packet */
          (*midisys) ( ),      /* system MIDI vector */
          (*ikbdsys) ( );      /* system ikbd vector */
}
```

The **midivec** vector is initialized to a buffering routine in the BIOS. The **vkbderr** and **vmiderr** routines are called whenever an overrun condition is detected on the keyboard or MIDI. The **statvec, mousevec, clockvec,** and **joyvec** point to the **ikbd** (intelligent keyboard) status, mouse, real-time clock, and joystick packet handlers. The **midisys** and **ikbdsys** routines are called when characters are available.

WORD **Kbrate (init, rpt)**
    WORD **init, rpt;**

Get or set the keyboard repeat rate. The number of 50 Hz ticks to wait before repeating is given by **init.** The number of 50 Hz ticks to wait between repeated characters is given by **rpt.** If either parameter is −1, that value is not changed.

The function returns the previous **init** and **rpt** values as a WORD with the high byte containing the **init** value and the low byte containing the **rpt** value.

long Kbshift (mode)
    WORD **mode;**

Get or set the keyboard shift bits. If **mode** is negative, the current settings are returned. If mode is 0 or greater, the shift bits are set to the low 8 bits of **mode.** The shift bit assignments are as follows:

Bit	*Key*
0	Right shift
1	Left shift
2	Control
3	Alternate
4	Caps Lock
5	Clr/Home (right mouse button)
6	Insert (left mouse button)
7	Reserved

long Keytbl (unshft, shft, capslock)
    char **unshft[ ], shft[ ], capslock[ ];**

Set the keyboard translation tables. Each parameter is a pointer to the base of an array of 128 characters. These tables are used to translate the keystrokes under the respective conditions. The function returns a pointer to a structure containing the three parameters:

```
struct keytab {
    char *unshift;
    char *shift;
    char *capslock;
}
```

long **Logbase( )**

Returns the address of the screen's logical location in memory immediately.

```
long Malloc (amtmem)
    long        amtmem;
```

Allocate **amtmem** bytes of memory. The function returns a pointer to a block of memory containing the requested number of bytes. A returned value of NULL indicates that there is no free block large enough to meet the request.

If **amtmem** is −1L, the function returns the size of the largest free block in the system.

A process may not have more than 20 blocks allocated by **Malloc( )** at any given time.

```
long Mediach (drv)
    WORD        drv;
```

Check for a media change on a disk. If the floppy disk has been replaced, the media change will be true. The values returned indicate the following:

0	Media definitely has not changed
1	Media might have changed
2	Media definitely has changed

The value of **drv** provides the drive number where 1 is A, 2 is B, etc.

```
WORD menu_bar (me_betree, me_bshow)
        OBJECT      *me_btree;
        WORD        me_bshow;
```

Displays a menu bar if **me_bshow** is TRUE and erases the menu bar if **me_bshow** is FALSE. The menu bar is the object tree pointed to by **me_btree.** An application should always erase the menu bar before exiting through **appl_exit( )**. A returned value of 0 indicates an error.

```
WORD menu_icheck (me_ctree, me_citem, mc_ccheck)
        OBJECT      *me_ctree;
        WORD        me_citem,
                    me_ccheck;
```

Check or uncheck a menu item. If **mc_check** is 1, the check mark is displayed in the first position of the menu item. If **mc_check** is 0, the

check mark is removed. Parameter **me_ctree** points to the object tree for the menu and **me_citem** is the object index in that tree. A returned value of 0 indicates an error.

```
WORD menu_ienable (me_etree, me_eitem, me_eenable)
        OBJECT      *me_etree;
        WORD        me_eitem,
                    me_eenable;
```

Enable or disable a menu item. If **me_eenable** is 0, the menu item is disabled and is displayed using dimmed characters. If **me_eenable** is 1, the menu item is enabled. Parameter **me_etree** points to the object tree for the menu and **me_eitem** is the object index of the menu item. A returned value of 0 indicates an error.

```
WORD menu_register (me_repid, me_rpstring)
        WORD        me_repid;
        char        *me_repstring;
```

Register a desk accessory with the AES. When a desk accessory is first executed, it must register with the AES to be shown in the Desk menu. The accessory uses this function to notify the AES that the application with ID number **me_repid** (from the **appl_init( )** function) is to be placed in the Desk menu with the name given in **me_rpstring.** The **menu_register( )** function returns a menu item ID number or −1. A −1 value indicates an error, usually meaning that six accessories are already in the Desk menu.

```
WORD menu_text (me_ttree, me_titem, me_ttext)
        OBJECT      *me_ttree;
        WORD        me_titem;
        char        *me_ttext;
```

Change the text of a menu item. Parameter **me_ttree** points to the object tree for the menu and **me_titem** is the object index of the menu item. The next text is pointed to by **me_ttext.** The new text must be no longer than the length of the original text in the menu. A 0 returned value indicates an error.

```
WORD menu_tnormal (me_ntree, me_ntitle, me_nnormal)
        OBJECT      *me_ntree;
        WORD        me_ntitle,
                    me_nnormal;
```

Set a menu title to normal or reverse video display. If **me_normal** is 0,

the menu title is displayed in reverse video. If **me_nnormal** is 1, the menu title is shown in normal video. Parameter **me_ntree** points to the object tree for the menu and **me_ntitle** is the object index of the menu title. A returned value of 0 indicates an error.

```
VOID Mfpint (interno, vecptr)
     WORD      interno,
               (*vecptr) ( );
```

Set the 68901 MFP (Multi-Function Peripheral) interrupt number interno to the new vector vecptr. The old routine vector is written over and unrecoverable. The interrupt numbers are:

Interrupt	*Function*
0	Parallel port (initially disabled)
1	RS232 carrier detect (initially disabled)
2	RS232 clear to send (initially disabled)
3	Unused, disabled
4	Unused, disabled
5	200 Hz system clock
6	Keyboard/MIDI (6850)
7	Polled floppy/hard disk controllers (initially disabled)
8	Hsync (initially disabled)
9	RS232 transmit error
10	RS232 transmit buffer empty
11	RS232 receive error
12	RS232 receive buffer empty
13	Unused, disabled
14	RS232 ring detect (initially disabled)
15	Polled monitor type (initially disabled)

```
WORD Mfree (saddr)
     long      saddr;
```

Free the block of memory starting at address **saddr.** The block of memory *must* have been previously allocated by **Malloc( ).** A value of 0 is returned if successful; otherwise an error number is returned.

```
VOID Midiws (cnt, ptr)
     WORD      cnt;
     char      *ptr;
```

Write a string to the MIDI port. Parameter cnt gives the number of characters to write, minus one. The string to write is pointed to by **ptr.**

```
WORD Mshrink (zero, baddr, newsiz)
        WORD    zero;
        long    baddr,
                newsiz;
```

Shrink the size of an allocated block of memory. The address of the block is given by **baddr.** The new size is specified by **newsiz.** The new size must be less than the current allocation. The parameter **zero** must have the value 0. The function returns 0 if successful, or an error number if not.

```
WORD objc_add (ob_atree, ob_aparent, ob_achild)
        OBJECT  *ob_atree;
        WORD    ob_aparent,
                ob_achild;
```

Add an object to an object tree. All object trees in the AES are stored as arrays. Each object in the tree is placed in an element in the array. The tree is created by linking the objects together. This function adds the object at index **ob_achild** to the list of children for the object at **ob_aparent.** The parameter **ob_atree** points to the object tree (base of the array). A 0 value returned indicates an error.

```
WORD objc_change (ob_ctree, ob_cobject, ob_cresvd,
            ob_cxclip, ob_cyclip, ob_cwclip, ob_chclip,
            ob_cnewstate, ob_credraw)
        OBJECT  *ob_ctree;
        WORD    ob_cobject,
                ob_cresvd,
                ob_cxclip, ob_cyclip, ob_cwclip, ob_chclip,
                ob_cnewstate, ob_credraw;
```

Change an object's **ob_state** field in the OBJECT structure (see text for object structures). The object index **ob_cobject** in the tree pointed to by **ob_ctree** has its **ob_state** field changed to the value in **ob_cnewstate.** The object is redrawn if **ob_credraw** is 1. If **ob_credraw** is 0, the object is not redrawn. The redraw operation is limited by the clipping rectangle indicated by **ob_cxclip, ob_cyclip, ob_cwclip,** and **ob_chclip** for the x and y coordinates, width, and height, respectively. A returned value of 0 indicates an error.

```
WORD objc_delete (ob_dltree, ob_dlobject)
        OBJECT  *ob_dltree;
        WORD    ob_dlobject;
```

Delete (unlink) an object from a tree. The tree is pointed to by **ob_dltree.** The starting object index is given by **ob_drstartob.** The number of objects to draw is given by **ob_dlobject.** A returned value of 0 indicates an error.

```
WORD objc_draw (ob_drtree, ob_drstartob, ob_drdepth,
        ob_drxclip, ob_dryclip, ob_drwclip, ob_drhclip)
        OBJECT  *ob_drtree;
        WORD    ob_drstartob,
                ob_drdepth,
                ob_drxclip, ob_dryclip,
                ob_drwclip, ob_drhclip;
```

Draw any object or objects in an object tree. The object tree containing the object(s) is pointed to by **ob_drtree.** The starting object index is given by **ob_drstartob.** The number of objects to draw is given by **b_drdepth,** which specifies the depth of levels to cover. Level 0 is the starting object, level 1 is the starting object's children, and so on. The depth specified may exceed the depth of the tree without incident.

All objects are drawn within a clipping rectangle. The upper left corner of clipping rectangle is given by **ob_drxclip** and **ob_dryclip.** The width and height are given **by ob_drwclip** and **ob_drhclip.** All coordinates are given as screen coordinates.

A returned value of 0 indicates an error.

```
WORD objc_edit (ob_edtree, ob_edobject, ob_edchar, ob_edidx,
            ob_edkind, ob_ednewidx)
        OBJECT  *ob_edtree;
        WORD    ob_edobject,
                ob_edchar,
                ob_edidx,
                ob_edkind,
                *ob_ednewidx;
```

Let the user edit the text of an object. The object index is **ob_edobject** in the tree pointed to by **ob_edtree.** The object *must* be of type G_TEXT or G_BOXTEXT (see text for descriptions of object types and object structures). The character entered is in **ob_edchar.** The position (index) of the next character in the text string is in **ob_edidx.** The edit function to perform is given by **ob_edkind** using the following codes:

---

0 Reserved
1 Combined values in the te_ptext and te_ptmplt fields into a formatted string and turn on the text cursor

2	Validate typed characters against the te_pvalid field, update the te_ptext field, and display the string
3	Turn off the text cursor

When the function is finished, **ob_ednewidx** contains the next character position in the text string after the edit function is complete. A returned value of 0 indicates an error.

```
WORD objc_find (ob_ftree, ob_fstartob, ob_fdepth, ob_fmx, ob_fmy)
        OBJECT  *ob_ftree;
                ob_fstartob,
                ob_fdepth,
                ob_fmx, ob_fmy;
```

Locate an object at a given x and y coordinate. The x and y coordinates are passed in **ob_fmx** and **ob_fmy**, and are screen coordinates. The tree to search is pointed to by **ob_ftree**. The object index to start searching from is given by **ob_fstartob**, and **ob_fdepth** limits the depth of the search. The function returns the object index of the object located at the specified position, or −1 if no object was found.

```
WORD objc_offset (ob_oftree, ob_ofobject, ob_ofxoff, ob_ofyoff)
        OBJECT  *ob_oftree;
        WORD    ob_ofobject,
                *ob_ofxoff, *ob_ofyoff;
```

Convert an object's relative coordinates to screen coordinates. The object index is passed in **ob_ofobject** and the tree is pointed to by **ob_oftree**. The screen coordinates are returned in **ob_ofxoff** and **ob_ofyoff**. A returned value of 0 indicates an error.

```
WORD objc_order (ob_ortree, ob_orobject, ob_ornewpos)
        OBJECT  *ob_ortree;
        WORD    ob_orobject,
                ob_ornewpos;
```

Change the order of an object within its parent's list of children. Parameter **ob_ortree** points to the object tree and **ob_orobject** is the index of the object to change. The new position is specified in **ob_ornewpos**. The value for the new position is:

−1	On the top
0	On the bottom (at the end)
1	One from the bottom
2	Two from the bottom
etc.	

VOID **Offgibit (bitno)**
    WORD **bitno;**

Turn off bit number **bitno** in the Port A register of the sound chip. This action is done without interruption by the operating system.

VOID **Ongibit (bitno)**
    WORD **bitno;**

Turn off bit number **bitno** in the Port A register of the sound chip. This action is done without interruption by the operating system.

long **Pexec (mode, path, cmdtail, environ)**
    WORD **mode;**
long **char** ***path, *cmdtail, *environ;**

This function performs several operations based upon the following value in mode:

Mode	*Operation*
0	Load and execute
3	Just load
4	Just execute
5	Create the base page

The **path** is the filename of the program to load. The **cmdtail** is the command line to be used by the program. The **environ** is the environment string to placed in the base page. If **environ** is 0L, the parent program's environment string is used.

If **mode** is 0, the file is loaded, the base page is created, and the program is executed. The function returns the exit status of the child process.

If **mode** is 3, the file is loaded and the base page is created. The function returns the address to the base page.

If **mode** is 4, path is the address of the base page and the program begins executing.

If **mode** is 5, the function allocates the largest free block of memory for the child process and creates most of the base page. The text, data, and bss size and the base values are *not* setup. The parent process is responsible for maintaining these values.

The child process will inherit the parent's standard file descriptors. Any **Fdup( )** or **Fforce( )** calls will be carried over to the child process.

long **Physbase()**

Returns the address of the screen's physical location in memory at the next vertical blank interrupt.

VOID **Protobt (buf, serialno, disktype, execflag)**
```
char    *buf;
long    serialno;
WORD    disktype, execflag;
```

Create a prototype boot sector. The boot sector to be created is placed in the 512 byte buffer pointed to by **buf.** The serial number stamped into the boot sector is given by **serialno.** If **serialno** is −1, the previous serial number is used. If **serialno** is greater than 0x01000000, a random serial number is used.

The disk type is given by **disktype** having the following interpretation:

0	One side, 180K, 40 tracks
1	Two sides, 360K, 40 tracks
2	One side, 360K, 80 tracks
3	Two sides, 720K, 80 tracks

If **disktype** is −1 and **buf** points to an existing boot sector, the disk type information remains unchanged.

The executable status of the boot sector is given by **execflag.** A 0 value means nonexecutable and a 1 value means executable. A −1 value with a valid boot sector in **buf** causes no change to the executable status.

VOID **Pterm (retcode)**
```
WORD    retcode;
```

Terminate the current process, close all open files, and release any allocated memory. The parameter **retcode** is the process's exit code that is returned to the parent process.

VOID **Pterm0()**

Terminate the current process with an exit status of 0, close all open files, and release all memory allocated to it. Source file must include OSBIND.H.

VOID **Ptermres (keepcnt, retcode)**
    long **keepcnt;**
    WORD **retcode;**

Terminate the current process and keep it in memory. The value in **keepcnt** determines the amount of the program to retain in memory. An memory requested and allocated by the program is *not* released. The value of **retcode** is the exit code returned to the parent process.

VOID **Puntaes ()**

This function causes the system to reboot without loading the AES or the GEM Desktop. If the system is in ROM (Read Only Memory), this function does not work. If this function had already been called, it simply returns to the calling program.

long **Random ()**

Return a 24-bit random number. The following algorithm is used:

S = (S * C) + K

where K is 1, C is 3141592621, and S is the seed. The initial value for S is taken from the system variable **_frclock**.

WORD **rc_intersect(p1, p2)**
    GRECT ***p1, *p2;**

Find the intersection of two rectangles. Each parameter refers to a GRECT structure which holds the values for an AES rectangle (x, y, width, and height). The function calculates the rectangle formed by the intersection of p1 and p2. The intersecting rectangle is returned through p2. The function returns TRUE if the rectangles intersect; otherwise FALSE is returned.

VOID **Rsconf (speed, flowctl, ucr, rsr, tsr, scr)**
    WORD **speed, flowctl, ucr, rsr, tsr, scr;**

Configure the RS232 port. The baud rate is set by speed as follows:

Speed	*Baud Rate*
0	19200
1	9600
2	4800

(*continued*)

Speed	*Baud Rate*
3	3600
4	2400
5	2000
6	1800
7	1200
8	600
9	300
10	200
11	150
12	134
13	110
14	75
15	50

The flow control is determined by **flowctl** as follows:

Flowctl	*Type of Flow Control*
0	No flow control used (default)
1	XON/XOFF
2	RTS/CTS
3	Both XON/XOFF and RTS/CTS

The remaining parameters set the hardware registers for the 68901 MFP (multifunction peripheral). Any parameter may be −1 to avoid setting the hardware register. Only the **ucr** parameter is useful. Its bits have the following meanings:

Bit in ucr	*Meaning*
0	Not used
1	Parity: 0 = odd 1 = even
2	Parity enable: 0 = off 1 = on
3 and 4	Number of start/stop bits:

Bit 4	Bit 3	Start	Stop	Format
0	0	0	0	Synchronous
0	1	1	1	Asynchronous
1	0	1	1.5	Asynchronous
1	1	1	2	Asynchronous

(*continued*)

Bit in ucr	*Meaning*
5 and 6	Word length: Bit 6 Bit 5 Word length 0 0 8 bits 0 1 7 bits 1 0 6 bits 1 1 5 bits
7	Clock mode: 0 = full speed 1 = 1/16 rate (recommended)

WORD **rsrc_free** ()

Free the memory space allocated to the resource file of an application. This function should be called prior to exiting the application. A returned value of 0 indicates an error.

```
WORD rsrc_gaddr (re_gtype, re_gindex, re_gaddr)
        WORD        re_gtype,
                    re_gindex;
        char        **re_gaddr;
```

Locate the address of the data structure requested. The address of the data structure is returned through parameter **re_gaddr**, which is a pointer to a pointer that references the appropriate data structure. The index of the data structure within the object tree is given by **re_gindex**. The type of structure to search for is given by **re_gtype** using the following predefined constants:

Constant	*Value*	*Structure*
R_TREE	0	tree
R_OBJECT	1	OBJECT
R_TEDINFO	2	TEDINFO
R_ICONBLK	3	ICONBLK
R_BITBLK	4	BITBLK
R_STRING	5	string
R_IMAGEDATA	6	image data
R_OBSPEC	7	ob_spec field of OBJECT
R_TEPTEXT	8	te_ptext field of TEDINFO
R_TEPTMPLT	9	te_ptmplt field of TEDINFO
R_TEPVALID	10	te_pvalid field of TEDINFO
R_IBPMASK	11	ib_pmask field of ICONBLK
R_IBPDATA	12	ib_pdata field of ICONBLK
R_IBPTEXT	13	ib_ptext field of ICONBLK
R_BIPDATA	14	bi_pdata field of BITBLK
R_FRSTR	15	address of pointer to a free string
F_FRIMG	16	address of pointer to a free image

The function returns a 0 if an error has occurred.

WORD **rsrc_load (re_lpfname)**
char ***re_lpfname**

Load a resource file into memory. This function loads the resource file named by **re_lpfname** into memory. The function allocates the memory, loads the file, adjusts the data to link all pointers, and makes the data available to the application. A returned value of 0 indicates an error.

WORD **rsrc_obfix (re_otree, re_oobject)**
OBJECT ***re_otree;**
WORD **re_oobject;**

Convert an object's location and size from character coordinates to pixel coordinates. The character coordinates are defined as the object's x, y, width, and height values. Each value is stored as a WORD which has a Least Significant Byte and a Most Significant Byte. The Least Significant Byte contains the character position of the value. The Most Significant Byte contains the positive or negative pixel offset from that character position.

The index of the object is given by **re_oobject** in the tree pointed to by **re_otree.** The function always returns the value 1.

WORD **rsrc_saddr (re_stypel, re_sindex, re_saddr)**
WORD **re_stype,**
re_sindex;
char ***re_saddr;**

Set the address of a data structure. The address of the data structure is given by **re_saddr** which may be a pointer to the data structure itself. The index at which the address is stored is given by **re_sindex.** The type of structure to store is given by **re_gtype** using the following predefined constants:

Constant	*Value*	*Structure*
R_TREE	0	tree
R_OBJECT	1	OBJECT
R_TEDINFO	2	TEDINFO
R_ICONBLK	3	ICONBLK
R_BITBLK	4	BITBLK
R_STRING	5	string
R_IMAGEDATA	6	image data
R_OBSPEC	7	ob_spec field of OBJECT
R_TEPTEXT	8	te_ptext field of TEDINFO

(*continued*)

Constant	*Value*	*Structure*
R_TEPTMPLT	9	te_ptmplt field of TEDINFO
R_TEPVALID	10	te_pvalid field of TEDINFO
R_IBPMASK	11	ib_pmask field of ICONBLK
R_IBPDATA	12	ib_pdata field of ICONBLK
R_IBPTEXT	13	ib_ptext field of ICONBLK
R_BIPDATA	14	bi_pdata field of BITBLK
R_FRSTR	15	address of pointer to a free string
F_FRIMG	16	address of pointer to a free image

The function returns a 0 if an error has occurred.

long **Rwabs (rwflag, buf, count, recno, drv)**
    WORD **rwflag;**
    char ***buf;**
    WORD **count, recno, drv;**

Read or write a logical sector on a device. This function returns 0 if successful; otherwise a negative error value is returned. The parameters are used as follows:

rwflag	determines the read/write mode as: 0 Read 1 Write 2 Read, but do not affect media change 3 Write, but do not affect media change
buf	points to the buffer to read or write (buf may be an odd address, but it will be slow)
count	is the number of sectors to transfer
recno	is the starting logical sector number
drv	is the drive device number where: 0 Floppy drive A 1 Floppy drive B >1 Hard disks, networks, or other devices

VOID **Scrdmp ()**

Transfer the screen image to the printer. This function only works with the monochrome monitor and Atari-compatible printers.

WORD **scrp_read (sc_rpscrap)**
    char ***sc_rpscrap;**

Read the current scrap directory on the clipboard. This function reads the scrap directory and places the data found into the buffer pointed to by **sc_rpscrap.** This function returns a 0 if an error has occurred.

WORD **scrp_write (sc_wpscrap)**
    char ***sc_wpscrap;**

Write to the current scrap directory. This function writes the data from the buffer pointed to by **sc_wpscrap** into the current scrap directory. Only one scrap directory file is available; therefore, writing to the scrap directory writes over an previous data located there. This function returns a 0 if an error condition exists.

long **Setexc (vecnum, vec)**
    WORD **vecnum;**
    WORD **(*vec) ();**

Change one of the exception vectors. The vector to change is given by **vecnum** and the address of the new routine is given by **vec.** If **vec** equals −1L, no change is made to the vector. The function returns the previous vector routine address, or −1 if the vector could not be set.

The vectors used by the ST are:

0x0000 to 0x00FF	Reserved for the 68000
0x0100	GEMDOS system timer interrupt
0x0101	GEMDOS critical error handler
0x0102	GEMDOS process termination
0x0103 to 0x0107	GEMDOS reserved
0x0200 and above	Reserved for OEM usage

Refer to the GEMDOS and Motorola 68000 documentation for further information about interrupts and vectors.

WORD **Setcolor (colornum, colorset)**
    WORD **colornum, colorset;**

Change the color of a single color palette entry. The palette entry to change is given by **colornum.** The new color is determined by the bit settings in **colorset** (see text). If **colorset** is negative, the entry is not changed. The function returns the previous color setting for the entry.

VOID **Setpalette (newpalette)**
    WORD ***newpalette;**

Set the Atari ST color palette. The parameter **newpalette** points to an array of 16 WORDs containing the 16 entries for the new color palette. The new palette is set at the next vertical blank interrupt.

WORD **Setprt (config)**
WORD **config;**

Sets the printer configuration to the bit settings in **config.** If **config** is −1, the settings are not changed. The function returns the previous printer configuration.

The bit settings have the following interpretation:

Bit	When = 0	When = 1
0	Dot matrix	Daisy wheel
1	Color	Monochrome
2	Atari printer	"Epson" printer
3	Draft mode	Final mode
4	Parallel port	RS232 port
5	Continuous form	Single sheet
6–14	Reserved	
15	Must always be 0	

VOID **Setscreen (log_loc, phys_loc, rez)**
char ***log_loc, *phys_loc;**
WORD **rez;**

Set the logical screen address, the physical screen address, and the physical screen resolution. Parameter **log_loc** contains the address of the new logical screen location, and **phys_loc** has the address of the new physical screen location. The resolution is set to the value in **rez.**

The logical screen location is set immediately, whereas the physical screen location does not change until the next vertical blank interrupt. When the resolution is changed, the screen is cleared, the cursor is put at the home position (upper left), and the VT52 terminal emulator is reset.

If any parameter has a negative value, that parameter is ignored. Therefore, it is possible to set any one or two values with this function.

VOID **Settime (datetime)**
long **datetime;**

Set the current date and time. The bits in **datetime** have the following interpretation:

Bits	*Meaning*
0–4	Seconds divided by 2 (0–29)
5–10	Minutes (0–59)
11–15	Hours (0–23)

(*continued*)

Bits	*Meaning*
16–20	Day in month (1–31)
21–24	Month (1–12)
25–31	Years since 1980 (0–119)

WORD **shel_envrn (sh_spvalue, sh_eparm)**
char ****sh_spvalue, *sh_eparm;**

Search in the environment settings for the occurrence of an environment parameter string. The string to search for is pointed to by **sh_eparm** (this includes the "=" character). The address of the byte immediately following the parameter string is returned through **sh_spvalue.** This function always returns the value 1.

WORD **shel_find (sh_fpbuff)**
char ***sh_fpbuff;**

Search for a filename in the current directory and in each directory in the search path. The filename to search for is given by **sh_fpbuff.** If the file is found, **sh_fpbuff** is returned with the full GEMDOS file specification. The buffer used to hold the file name must be at least 80 characters long. The returned value of 0 indicates an error.

WORD **shel_read (sh_rpcmd, sh_rptail)**
char ***sh_rpcmd, *sh_rptail;**

Let an application identify the command that invoked it. The command used to invoke the application is pointed to by **sh_rpcmd.** The command tail buffer is pointed to by **sh_rptail.** The command tail buffer has the following format:

WORD	Address of environment string
pointer	To command line
pointer	To default File Control Block
pointer	To second default File Control Block

The function returns a 0 if an error occurred.

WORD **shel_write (sh_wdoex, sh_wisgr, sh_wiscr,**
sh_wpcmd, sh_wptail)
WORD **sh_wdoex, sh_wisgr, sh_wiscr;**
char ***sh_wpcmd, *sh_wptail;**

Tell GEM AES whether to run another application. The parameter **sh_wdoex** instructs the AES to exit to the Desktop or run another application when the user exits the current application. The following flags are used:

0 = exit to Desktop
1 = run another application

Parameter **sh_wisgr** tells the AES whether the next application is a graphic application as follows:

0 = not graphic application
1 = graphic application

Parameter **sh_wiscr** tells the AES whether the next application is a GEM application as follows:

0 = not GEM application
1 = GEM application

The parameters **sh_wpcmd** and **sh_wptail** provide the name of the next application and its command tail, respectively.

The function returns a 0 if an error has occurred.

long **Super (stackptr)**
    WORD ***stackptr;**

Change the supervisor mode of the 68000 processory. If **stackptr** equals −1L, the function is inquiring about the status of the processor. A returned value of 0 means the processor is in user mode, and a value of 1 means the processor is in supervisor mode.

Otherwise the function toggles the processor between user and supervisor modes. If the processor is in user mode, the processor is put into supervisor mode and the new stack is at the address in parameter **stackptr.** If **stack** equals 0, the supervisor stack location remains the same.

If the processor is in supervisor mode, it is placed into user mode and the stack location changes to the address held in parameter **stackptr** (or remains the same if stack equal 0). When returning to user mode, the stack location should be reset to the location used prior to entering supervisor mode. The stack location *must* be restored to its original location before the process terminates.

VOID **Supexec (codeptr)**
    WORD **(*codeptr) ();**

Quick entry to supervisor mode. The function places the 68000 into supervisor mode and begins executing the code at location **codeptr.** When the routine at **codeptr** returns, the 68000 is put back to user mode, and the program continues from the **Supexec( )** call.

Note: the code at **codeptr** cannot perform BIOS or GEMDOS calls. This function is useful for quick changes to protected memory locations and hardware functions.

WORD **Sversion ( )**

Get the version number of GEMDOS. The high byte contains the minor version number and the low byte the major version number.

WORD **Tgetdate ( )**

Return the current date in the following format:

Bits	*Data*
0–4	The day ranging from 1 to 31
5–8	The month ranging from 1 to 12
9–15	The year since 1980 ranging from 0 to 119

WORD **Tgettime ( )**

Return the current time of day in the following format:

Bits	*Data*
0–4	Seconds divided by 2 ranging from 0 to 29
5–10	Minutes ranging from 0 to 59
11–15	Hours ranging from 0 to 23

long **Tickcal( )**

Return the system timer calibration value to the nearest millisecond. Generally a useful function because the number of elapsed milliseconds is passed on the stack when a system timer exception occurs.

WORD **Tsetdate (newdate)**
    WORD **newdate;**

Set the current date. The value returned is 0 if GEMDOS accepted the date (although February 31 may be accepted); otherwise an error value is returned. The parameter newdate follows the format described in function **Tgetdate( )**.

```
WORD Tsettime (newtime)
        WORD        newtime;
```

Set the current time. A value of 0 is returned if GEMDOS accepted the time; otherwise an error value is returned. The format of parameter newtime is the same as that described in **Tgettime( )**.

```
WORD v_arc (handle, x, y, radius, begang, endang)
        WORD        handle,
                    x, y, rad,
                    begang, endang;
```

Draw a circular arc. The center of the circle is placed at coordinates (x,y) and the radius, rad, is measured along the x axis. The arc starts at the angle **begang** and proceeds counterclockwise to angle **endang.** The parameters measure angles in tenths of a degree and can range from 0 through 3600.

```
WORD v_bar (handle, pxyarray)
        WORD        handle,
                    pxyarray[4];
```

Draw a filled rectangular area. The rectangle is given by array **pxyarray.** The current fill area attributes are used. As opposed to the **vr_recfl( )** function, the perimeter is drawn based upon its current setting.

```
WORD v_bit_image (handle, filename, paspect, x_scale, y_scale,
            h_align, v_align, xyarray)
        WORD        handle;
        char        *filename;
        WORD        paspect,
                    x_scale, y_scale,
                    h_align, v_align,
                    xyarray;
```

Output bit image file. This function allows an application to print a bit image file (file type . IMG) on the printer. The file to print is specified by its file name in string **filename.**

The bit image file contains the pixel sizes, and the printer driver is able to calculate a pixel aspect ratio. Using the aspect ratio means that

images retain the vertical and horizontal relationship that appeared on the original device. For example, a circle on the original device appears as a circle on the printer. If the aspect ratio is not used, the relationship may not be retained. Parameter **paspect** controls the use of the aspect ratio, where a value of 1 means retain the ratio, and 0 means ignore the ratio.

The size of the x and y axes are probably not the same on the printer as they were on the original device. This function allows the application to specify whether fractional or integral scaling are used. Fractional scaling ensures that the corresponding axis fits exactly within the scaling rectangle. Integral scaling does not guarantee this, but is generally faster. The parameters **x_scale** and **v_scale** determine the type of scaling for each axis. A value of 1 indicates integral and 0 indicates fractional scaling.

Parameters **h_align** and **v_align** determine horizontal and vertical alignment as follows:

```
h_align = 0: left
          1: center
          2: right
v_align = 0: top
          1: middle
          2: bottom
```

The scaling rectangle is optional. It is specified through array **xyarray.** The scaled bit image always resides within the scaling rectangle. If a combination of preserved pixel aspect ratio, scaling, or alignment causes the scaled bit image to extend beyond the edge of the scaling rectangle, the VDI clips the bit image to that edge.

```
WORD v_cellarray (handle, pxyarray, row_length, el_used,
          num_rows, wrt_mode, colarray)
     WORD      handle,
               pxyarray[4],
               row_length, el_used, num_rows, wrt_mode,
               colarray[num_rows * el_used];
```

Draw a cell array. The function draws a rectangular array within the rectangle given by **pxyarray.** The rectangle is divided into cells based upon the number of rows, **num_rows,** and number of columns, **row_length.** The color index array, **colarray,** determines the color of each cell. The parameter **el_used** determines the number of elements per row in **colarray.**

If the device does not support cell arrays, the device outlines the area with a solid line. This function is not required and may not be available on all devices.

WORD **v_circle (handle, x, y, rad)**
WORD **handle,**
x, y, rad;

Draw a circle. The center of the circle is a coordinate (**x,y**). The radius, **rad,** of the circle is measured along the x axis. Fill area attributes are used.

WORD **v_clear_disp_list (handle)**
WORD **handle;**

Clear display list. This function clears the output display list for the printer. It is similar to the **v_clrwk( )** function except that a form advance is not incurred.

WORD **v_clrwk (handle)**
WORD **handle;**

Clear the workstation specified by **handle.**

WORD **v_clsvwk (handle)**
WORD **handle;**

Close the virtual workstation specified by the device handle in parameter **handle.**

WORD **v_clswk (handle)**
WORD **handle;**

Close a physical workstation with the handle specified in parameter **handle.** This function performs all necessary actions to complete output to the device. For example, a printer device is updated, a metafile is closed, and a screen is put into alpha mode. Remember to close all open workstations associated with the physical workstation before closing the physical workstation.

WORD **v_contourfill (handle, x, y, index)**
WORD **handle,**
x, y,
index;

Fill an area on the display. Beginning at the starting point (**x,y**), the function fills the area using the current fill attributes. The fill proceeds to the edge of the display surface, or to a pixel with the color index given by **index.** If **index** is negative, any color other than color of

the pixel at (**x,y**) becomes a boundary of the fill. This function is not required and may not be available on all devices.

WORD **v_curdown (handle)**
    WORD **handle;**

Alpha cursor down. Move the alpha cursor down one row without changing its horizontal position. If the cursor is at the bottom row, nothing happens.

WORD **v_curhome (handle)**
    WORD **handle;**

Home alpha cursor. Move the alpha cursor to its home position, usually the upper left corner.

WORD **v_curleft (handle)**
    WORD **handle;**

Alpha cursor left. Move the alpha cursor one column to the left without changing its vertical position. If the cursor is at the furthest left column, nothing happens.

WORD **v_curright (handle)**
    WORD **handle;**

Alpha cursor right. Move the alpha cursor one column to the right without changing its vertical position. If the cursor is at the furthest right column, nothing happens.

WORD **v_curtext (handle, str)**
    WORD **handle;**
    char ***str;**

Output cursor addressable alpha text. This function writes the string in **str** starting at the current alpha cursor position. The alpha text attributes currently in effect are used for the output.

WORD **v_curup (handle)**
    WORD **handle;**

Alpha cursor up. Move the alpha cursor up one row without changing its horizontal position. If the cursor is at the top row, nothing happens.

```
WORD v_dspcur (handle, x, y)
        WORD    handle,
                x, y;
```

Place graphic cursor. The graphic cursor is placed at the location specified by **x** and **y**.

```
WORD v_eeol (handle)
        WORD    handle;
```

Erase to end of alpha text line. This function erases all alpha cells from the current alpha cursor position to the end of the current line. The cell at the current alpha cursor position is not changed.

```
WORD v_eeos (handle)
        WORD    handle;
```

Erase to end of alpha screen. This function erases all alpha cells from the current alpha cursor position to the end of the screen. The cell at the current alpha cursor position is not changed.

```
WORD v_ellarc (handle, x, y, xrad, yrad, begang, endang)
        WORD    handle,
                x, y,
                xrad, yrad,
                begang, endang;
```

Draw an ellipse. The center of the ellipse is placed at coordinate (**x**,**y**). The x radius is given by **xrad** and is measured along the x axis from the center. The y radius is measured along the y axis from the center and is given by **yrad**. The arc starts at angle **begang** and moves counterclockwise to **endang**. The angles measure tenths of degrees and range from 0 to 3600.

```
WORD v_ellipse (handle, x, y, xrad, yrad)
        WORD    handle,
                x, y,
                xrad, yrad;
```

Draw an ellipse. The center of the ellipse is placed at coordinate (**x**,**y**). The x radius is given by **xrad** and is measured along the x axis from the center. The y radius is measured along the y axis from the center and is given by **yrad**. Fill area attributes are used.

```
WORD v_ellpie (handle, x, y, xrad, yrad, begang, endang)
        WORD    handle,
                x, y,
                xrad, yrad,
                begang, endang;
```

Draw an ellipse. The center of the ellipse is placed at coordinate (**x**,**y**). The x radius is given by **xrad** and is measured along the x axis from the center. The y radius is measured along the y axis from the center and is given by **yrad.** The arc starts at angle **begang** and moves counterclockwise to **endang.** The angles measure tenths of degrees and range from 0 to 3600. After the arc is drawn, each end is connected to the center to complete the pie shape. Fill area attributes are used.

```
WORD v_enter_cur (handle)
        WORD    handle;
```

Enter alpha mode. This function causes the device to enter alpha mode if alpha mode is different from graphics mode. Using this function allows cursor address and ensures that the device makes the transition from graphics mode to alpha mode properly.

```
WORD vex_butv (handle, pusrcode, psavcode)
        WORD    handle,
                *pusrcode,
                **psavcode;
```

Exchange mouse button change vector. This function allows the application to specify a routine to be executed each time the state of the mouse button changes. The application routine receives control after the button state is determined and before the mouse button driver is activated. The application code address is given by **pusrcode.** The previous address is returned through **psavcode.**

When the application code is invoked, interrupts are disabled and should *not* be enabled. It is the responsibility of the application code to save and restore any registers it uses. The application code is started using the JSR instruction, and the routine should exit using the RTS instruction.

When the application routine is called, register D0.w contains the mouse button state. Each bit set to 1 indicates the button is pressed. Bit 0 corresponds to the furthest left button, bit 1 to the next button to the right, and so on. The application routine may change D0.w to force certain buttons to be down or up.

```
WORD vex_curv (handle, pusrcode, psavcode)
        WORD    handle,
                *pusrcode,
                **psavcode;
```

Exchange cursor change vector. This function allows the application to specify a routine to be executed each time the cursor is drawn. The application routine receives control whenever the cursor position should be updated. The application routine can take over drawing the cursor or can perform some action and have the VDI draw the cursor. The application code address is given by **pusrcode.** The previous address is returned through **psavcode.**

When the application code is invoked, interrupts are disabled and should *not* be enabled. It is the responsibility of the application code to save and restore any registers it uses. The application code is started using the JSR instruction, and the routine should exit using the RTS instruction.

When the application routine is called, register D0.w contains the new x coordinate and D1.w contains the new y coordinate. The application routine may change D0.w and D1.w to change the location of the cursor on the screen. If the application routine does not draw the cursor, it should issue a JSR instruction to the address in **psavcode** to draw the cursor.

```
WORD v_exit_cur (handle)
        WORD    handle;
```

Exit alpha mode. This function causes the device to enter graphics mode if graphics mode is different than alpha mode. Using this function ensures that the device makes the transition from alpha mode to graphics mode properly.

```
WORD vex_motv (handle, pusrcode, psavcode)
        WORD    handle,
                *pusrcode,
                **psavcode;
```

Exchange mouse movement vector. This function allows the application to specify a routine to be executed each time the mouse moves to a new location. The application routine receives control after the x and y coordinates are determined and before the mouse driver is updated or the mouse form is redrawn on the screen. The application code address is given by **pusrcode.** The previous address is returned through **psavcode.**

When the application code is invoked, interrupts are disabled and

should *not* be enabled. It is the responsibility of the application code to save and restore any registers it uses. The application code is started using the JSR instruction, and the routine should exit using the RTS instruction.

When the application routine is called, register D0.w contains the new x coordinate and D1.w contains the new y coordinate. The application routine may change D0.w and D1.w to change the location of the mouse on the screen.

```
WORD vex_timv (handle, tim_addr, otim_addr, tim_conv)
        WORD    handle,
                *tim_addr,
                **otim_addr,
                *tim_conv;
```

Exchange the timer interrupt vector. This function allows the application to execute a routine each time a timer tick occurs. The routine to be executed is given by **tim_addr.** The previous timer interrupt vector address is returned in **otim_addr.** The number of milliseconds per timer tick is returned in **tim_conv.**

When the application code is invoked, interrupts are disabled and should *not* be enabled. It is the responsibility of the application code to save and restore any registers it uses. The application code is started using the JSR instruction, and the routine should exit using the RTS instruction.

```
WORD v_fillarea (handle, count, pxyarray)
        WORD    handle,
                count,
                pxyarray[2 * count];
```

Fill a complex polygon. The number of vertices in the polygon is given by **count,** and the coordinate of the vertices are in **pxyarray.** The area is outlined with a solid line if the fill perimeter visibility is on (see **vsf_perimeter( )**). The area is filled using the attributes fill area color, interior style, writing mode, and style index.

If the device does not have area fill capability, the VDI draws the outline of the polygon using the current fill area color. The device drive ensures that the fill area is closed by connecting the last point to the first point.

A polygon with zero area is displayed as a dot if the perimeter visibility is on; otherwise nothing is drawn. A polygon with only one endpoint is ignored.

```
WORD v_form_adv (handle)
        WORD        handle;
```

Form advance. The printer advances to the next page. This function could be used instead of the **v_clrwk( )** function so that the current printer display list is retained.

```
WORD v_get_pixel (handle, x, y, pel, cindex)
        WORD        handle,
                    x, y,
                    *pel, *cindex;
```

Get the pixel value and color index. This function returns the pixel value and color index for the pixel at location **(x,y)**. The pixel value is returned in **pel** and the color index is returned in **cindex**.

Note that color index 0 is the background color. The VDI may return 0, or may return the index of the current color used for the background.

```
WORD v_gtext (handle, x, y, str)
        WORD        handle,
                    x, y;
        char        *str;
```

Write graphic text at the coordinates given by **x** and **y**. The text for the string to draw is pointed to by **str**. The text alignment is set by **vst_alignment( )**.

```
WORD v_hardcopy (handle)
        WORD        handle;
```

Hard copy. This function causes the image on the physical screen to be copied to a printer or other attached hard copy device. This function is device-specific and may not be available.

```
WORD v_hide_c (handle)
        WORD        handle;
```

Hide cursor. This function hides the current cursor form. The **v_show_c( )** and **v_hide_c( )** functions are nested. The **v_show_c( )** function must be called as many times as the **v_hide_c( )** function was called since the cursor disappeared for the cursor to become visible again. For example, if the cursor is visible and **v_hide_c( )** is called five times, **v_show_c( )** must be called five times before the cursor reappears. This nesting can be overridden through the **v_show_c( )** function.

```
WORD v_justified (handle, x, y, str, len, word_space, char_space)
        WORD    handle;
                x, y;
        char    *str;
        WORD    len, word_space, char_space;
```

Output justified text. The string to output is pointed to by **str.** The position of output is at coordinate (x,y). The length in which the justification is to take place is given by **len,** measured in the current coordinate system. If **word_space** is TRUE, inter-word spacing modification is used. If **char_space** is TRUE, inter-character spacing modification is used. The function uses the current text attributes.

```
WORD v_meta_extents (handle, min_x, min_y, max_x, max_y)
        WORD    handle,
                min_x, min_y,
                max_x, max_y;
```

Update metafile extents. This function writes the minimum and maximum x and y values to the metafile header. These extents provide an application with a quick indication of the minimum rectangle that will bound all primitives output to the metafile. If this function is not used when outputting to a metafile, the extents will be 0.

```
WORD vm_filename (handle, filename)
        WORD    handle;
        char    *filename;
```

Change the GEM VDI file name. The default file name for a metafile is GEMFILE.GEM. The new file name to be used is given by string **filename.** Only the filename portion of the string is used. The file type .GEM is always used. This function *must* be called immediately after opening the metafile with the **v_opnwk( )** function; otherwise the file name is not changed. The function also closes any open metafiles.

```
WORD v_opnvwk (work_in, handle, work_out)
        WORD    work_in[ ],
                *handle,
                work_out;
```

Open a virtual workstation. The values for arrays **work_in[ ]** and **work_out[ ]** are the same as for **v_opnwk( ).** The parameter **handle** has the device handle of the physical workstation when the function is called. Upon return, **handle** contains the new device handle for the virtual workstation. A returned device handle of 0 indicates a failure to open the virtual workstation.

Note: Not all input devices associated with the virtual workstation work.

```
WORD v_opnwk (work_in, handle, work_out)
        WORD        work_in[ ],
                    *handle,
                    work_out[ ];
```

Open a physical workstation for use by a program. This function uses the parameters in array **work_in[ ]** to set the initial workstation attributes. The new workstation handle is returned in parameter **handle**, and various workstation attributes are returned in array **work_out[ ]**. The function returns 0 as the device handle if a workstation could not be opened. The **work_in[ ]** array is defined:

Element	*Description*
0	Device id number indicating the device driver to be loaded as specified in file ASSIGN.SYS
1	Line type
2	Poly-line color index
3	Marker type
4	Poly-marker color index
5	Text face
6	Text color index
7	Fill interior style
8	Fill style index
9	Fill color index
10	Coordinate system selection 0 = Map full NDC to full RC 1 = Reserved 2 = Use RC system

The **work_out( )** array is defined:

Element	*Description*
0	Maximum addressable width in rasters or steps (that is, 640 means addressable area is from 0 through 639)
1	Addressable height in raster or steps
2	Device Coordinate units flag 0 = device is capable of precise scaling (usually a plotter or printer) 1 = device is not capable of precise scaling
3	Width of one pixel (or output unit) in microns
4	Height of one pixel in microns

Element	*Description*
5	Number of character heights; 0 means continuous scaling
6	Number of line types
7	Number of line widths; 0 means continuous scaling
8	Number of marker types
9	Number of marker sizes; 0 means continuous scaling
10	Number of faces supported (not the highest face number index)
11	Number of patterns
12	Number of hatch styles
13	The number of predefined colors that can be simultaneously displayed on the device
14	Number of generalized drawing primitives (GDPs)
15–24	These 10 elements indicate which GDPs are supported. Each GDP is given a value in the table below. If fewer than 10 GDPs are available, the value −1 is used to fill the remaining elements. 1 Bar 2 Arc 3 Pie slice 4 Circle 5 Ellipse 6 Elliptical arc 7 Elliptical pie 8 Rounded rectangle 9 Filled, rounded rectangle 10 Justified graphics text
25–34	For each GDP listed above, the corresponding element in this list indicates the attribute set for that particular GDP. The attribute values are: 0 Poly-line 1 Poly-marker 2 Text 3 Fill area 4 None
35	Color capability flag 0 = No; 1 = Yes
36	Text rotation capability flag 0 = No; 1 = Yes
37	Fill area capability flag 0 = No; 1 = Yes
38	Cell array operation capability flag 0 = No; 1 = Yes
39	Number of available colors (total number of colors in the color palette) 0 = Continuous (more than 32,767 colors) 2 = Monochrome (black and white) >2 = Number of colors
40	Number of locator devices 1 = Keyboard only 2 = Keyboard plus other devices
41	Number of valuator devices 1 = Keyboard only 2 = Other device available

Element	*Description*
42	Number of choice (button) devices 1 = Keyboard only 2 = Other device available
43	Number of string devices available 1 = Keyboard 2 = Other device available
44	Workstation type 0 = Output only 1 = Input only 2 = Input and output 3 = Reserved 4 = Metafile output
45	Minimum character width (not cell width) in current x coordinates
46	Minimum character character height in current y coordinates
47	Maximum character width in current x coordinates
48	Maximum charcter height in current y coordinates
49	Minimum line width in current x coordinates
50	0
51	Maximum line width in current x coordinates
52	0
53	Minimum marker width in current x coordinates
54	Minimum marker height in current y coordinates
55	Maximum marker width in current x coordinates
56	Minimum marker height in current y coordinates

```
WORD v_output_window (handle, xyarray)
        WORD        handle;
                    xyarray[4];
```

Output window. Output the rectangular area specified in **xyarray** to the printer. This function is similar to the **v_updwk( )** function except that the output rectangle must be specified.

```
WORD v_pieslice (handle, x, y, rad, begang, endang)
        WORD        handle,
                    x, y, rad,
                    begang, endang;
```

Draw a circular pie slice. The center of the circle is placed at coordinates (x,y) and the radius, **rad**, is measured along the x axis. The arc starts at the angle **begang** and proceeds counterclockwise to angle **endang**. The parameters measure angles in tenths of a degree and can range from 0 through 3600. The arc is drawn and each end is connected to the center to complete the slice. Fill area attributes are used.

```
WORD v_pline (handle, count, pxyarray)
        WORD    handle,
                count,
                pxyarray[2 * count];
```

Draw multiple line segments. The number of points in the **pxyarray** is given by **count.** The function starts drawing from the first point in the **pxyarray** and continues to connect points with line segments. The VDI displays a zero length line as a point. The attributes of color, line type, line width, end style, and current writing mode are all used.

```
WORD v_pmarker (handle, count, pxyarray)
        WORD    handle,
                count,
                pxyarray[2 *count];
```

Draw a set of markers at the points given in the **pxyarray.** The number of points to draw is given by **count.** The markers use the attributes color, scale, type, and writing mode.

```
WORD vq_cellarray (handle, pxyarray, row_length, num_rows,
            el_used, rows_used, stat, colarray)
        WORD    handle,
                pxyarray[4],
                row_length, num_rows,
                *el_used, *rows_used,
                *stat,
                colarray[row_length * num_rows];
```

Inquire cell array. This function returns the cell array definition of the pixels contained in the rectangle specified in **pxyarray.** Color indices are returned one row at a time in **colarray,** starting from the top of the rectangular area and proceeding downward. Parameter **row_length** is the length of each row in **colarray,** and **num_rows** is the number of rows in **colarray.**

The number of elements used in each row of **colarray** is returned in **el_used,** and the number of rows used in **colarray** is returned in **rows_used.** If an invalid value is found, stat is set to 1; otherwise **stat** returns as 0.

```
WORD vq_chcells (handle, num_rows, num_columns)
        WORD    handle,
                *num_rows, *num_columns;
```

Inquire addressable alpha character cells. This function returns the number of vertical (**num_rows**) and horizontal (**num_columns**) positions at which the alpha cursor can be positioned. If such addressing is not possible, the value is −1.

```
WORD vq_color (handle, color_index, set_flag, rgb)
        WORD    handle,
                color_index, set_flag,
                rgb[3];
```

Inquire color representation. This function returns the color output for color index **color_index**. The intensity for each color is returned in **rgb** where **rgb[0]** is red, **rgb[1]** is green, and **rgb[2]** is blue. The intensity ranges from 0 to 1000. The function can return the set or realized color intensities. The set color intensities are the intensities in the lookup table. The realized color intensities are the intensities used by the device (since a device may not have 1000 levels of intensity). The set values are returned when **set_flag** is 0. If **set_flag** is 1, the realized values are returned.

```
WORD vq_curaddress (handle, row_num, col_num)
        WORD    handle,
                *row_num, *col_num;
```

Inquire current alpha cursor address. The current row and column position of the alpha cursor is returned in **row_num** and **col_num**, respectively.

```
WORD vq_extnd (handle, owflag, work_out)
        WORD    handle,
                owflag,
                work_out[57];
```

Extended inquire. This function returns device-specific information regarding a workstation. This function has two output options. When **owflag** is 0, the **work_out** array contains the same values returned by the **v_opnwk( )** function. When **owflag** is 1, the **work_out** array elements are defined:

Element	*Description*
0	Screen type: 0 = not screen 1 = separate alpha and graphic controllers; separate video screens 2 = separate alpha and graphic controllers; common video screen

Element	*Description*
	3 = common alpha and graphic controllers; separate image memory
	4 = common alpha and graphic controllers; common image memory
1	Number of background colors available
2	Number of text effects (see vst_effects())
3	Scaling of rasters: 0 = no, 1 = yes
4	Number of planes
5	Lookup table: 0 = not supported, 1 = supported
6	Number of 16-by-16 pixel raster operations per second
7	Contour fill capability
8	Character rotation:
	0 = none
	1 = 90 degree increments only
	2 = arbitrary angles
9	Number of writing modes available
10	Input modes available:
	0 = none
	1 = request only
	2 = sample and request
11	Text alignment capability: 0 = no, 1 = yes
12	Inking capability: 0 = no, 1 = yes
13	Rubberbanding:
	0 = none
	1 = rubberband lines
	2 = rubberband lines and rectangles
14	Maximum number of vertices for poly-lines, poly-markers, or filled areas, −1 if there is no limit
15	Maximum integer inputs to VDI, or −1 if there is no limit
16	Number of available mouse keys
17	Line style capability for wide lines: 0 = no, 1 = yes
18	Writing modes for side lines
19–56	Reserved, all 0s

WORD **vqf_attributes (handle, attrib)**
    WORD **handle,**
    atrrib[4];

Inquire current fill area attributes. The function returns the current attribute settings for fill area operations in **attrib.** The elements of **attrib** are defined:

Element	*Description*
0	Fill interior style
1	Fill area color index
2	Fill area style index
3	Current writing mode

See also **vsf_color( )**, **vsf_inferior( )**, **vsf_perimeter( )**, **vsf_style( )**, and **vswr_mode( )**.

```
WORD vqin_mode (handle, dev_type, input_mode)
        WORD    handle,
                dev_type,
                *input_mode;
```

Inquire input mode. This function returns the current input mode (through **input_mode**) for the specified device type (**dev_type**). Valid device types are:

1 = locator
2 = valuator
3 = choice
4 = string

The input mode value returned is 1 for request mode and 2 for sample mode.

```
WORD vq_key_s (handle, pstatus)
        WORD    handle,
                *pstatus;
```

Sample keyboard state information. The function returns the current state of the keyboard's Control, Shift, and Alternate keys. The state of each key is returned as a bit setting in **pstatus**. If the bit is 1, the key is down. The bits are defined:

Bit	*Key*
0	Right shift
1	Left shift
2	Control
3	Alternate

```
WORD vql_attributes (handle, attrib)
        WORD    handle,
                attrib[4];
```

Inquire current poly-line attributes. The function returns the current attribute settings for poly-line operations in **attrib**. The elements of **attrib** are defined:

Element	*Description*
0	Current poly-line line type
1	Current poly-line color index
2	Current writing mode
3	Current line width

See also **vsl_color( )**, **vsl_ends( )**, **vsl_type( )**, **vsl_width( )**, and **vswr_mode( )**.

WORD **vqm_attributes (handle, attrib)**
    WORD **handle,**
    attrib[4];

Inquire current poly-marker attributes. The function returns the current attribute settings for poly-marker operations in **attrib**. The elements of **attrib** are defined:

Element	*Description*
0	Current poly-marker type
1	Current poly-marker color index
2	Current writing mode
3	Current marker height

See also **vsm_color( )**, **vsm_height( )**, **vsm_type( )**, and **vwsr_mode( )**.

WORD **vq_mouse (handle, pstatus, x, y)**
    WORD **handle,**
    ***pstatus,**
    ***x, *y;**

Sample mouse button state. The status of the mouse button is returned in bit settings of **pstatus**. If the bit is 1, the button is down. Bit 0 corresponds to the furthest left button, bit 1 to the next button to the right, and so on. The function also returns the current coordinates of the cursor in **x** and **y**.

WORD **vq_tabstatus (handle)**
    WORD **handle;**

Inquire tablet status. This function returns the availability status of a graphics tablet, mouse, joystick, or similar device. If the function returns 0, no tablet is available. If it returns 1, a tablet is available.

```
WORD vqt_attributes (handle, attrib)
        WORD    handle,
                attrib[0];
```

Inquire current graphic text attributes. The function returns the current attribute settings for graphic text operations in **attrib**. The elements of **attrib** are defined:

Element	*Description*
0	Graphic text face
1	Graphic text color index
2	Angle of text baseline rotation (0–3600)
3	Horizontal alignment
4	Vertical alignment
5	Writing mode
6	Character width in coordinate units
7	Character height in coordinate units
8	Cell width in coordinate units
9	Cell height in coordinate units

See also **vst_alignment( )**, **vst_color( )**, **vst_font( )**, **vst_height( )**, **vst_rotation( )**, and **vswr_mode( )**.

```
WORD vqt_extent (handle, str, extent)
        WORD    handle,
                extent[8],
        char    *str;
```

Inquire text extent. Given the string in **str**, the function returns four points describing a rectangle that encloses the text. The first point (**extent[0]** and **extent[1]**) corresponds to the lower left corner. The second point is the lower right corner. The third point is the upper right corner. The fourth point is the upper left corner. All text attributes, including effects and baseline rotation, affect the calculation.

```
WORD vqt_fontinfo (handle, minADE, maxADE,
            distances, maxwidth, fx)
        WORD    handle,
                *minADE, *maxADE,
                distances[5],
                *maxwidth,
                fx[3];
```

Inquire current face information. This function obtains size information regarding the current type face, taking into account the current height and effects. The ASCII Decimal Equivalent of the first and last characters in the font are returned in **minADE** and **maxADE**, respectively. The maximum cell width, not including effects, is returned through maxwidth. The distances array is defined:

Element	*Description*
0	Bottom line distance relative to baseline
1	Descent line distance relative to baseline
2	Half line distance relative to baseline
3	Ascent line distance relative to baseline
4	Top line distance relative to baseline

The **fx** array is defined:

Element	*Description*
0	Increase in character width due to effects
1	Left offset
2	Right offset

```
WORD vqt_name (handle, element_num, face_name)
        WORD        handle,
                    element_num;
        char        face_name[32];
```

Inquire face name and index. The function obtains the description data regarding face element **element_num**. The string **face_name** is set by the function and consists of two parts. The first 16 characters are the face name (e.g., Swiss 721 or Dutch 801), and the next 16 characters are the face style (e.g., Bold, Italic, Roman, etc.). The function returns the ID number of the current type face.

```
WORD vqt_width (handle, char, cell_width, left_delta, right_delta)
        WORD    handle,
                char,
                *cell_width, *left_delta, *right_delta;
```

Inquire character cell width. This function returns the character cell values for a particular character. The ASCII character code is given by

char. The cell width and left and right delta alignment are returned in **cell_widht, left_delta,** and **right_delta,** respectively (see Figure A-1). The function returns the value of **char,** or −1 if an invalid **char** value was used.

```
WORD v_rbox (handle, pxyarray)
        WORD    handle,
                pxyarray[4];
```

Draw a rectangle with rounded corners. The rectangle is given by the points in **pxyarray.**

```
WORD v_rfbox (handle, pxyarray)
        WORD    handle,
                pxyarray[4];
```

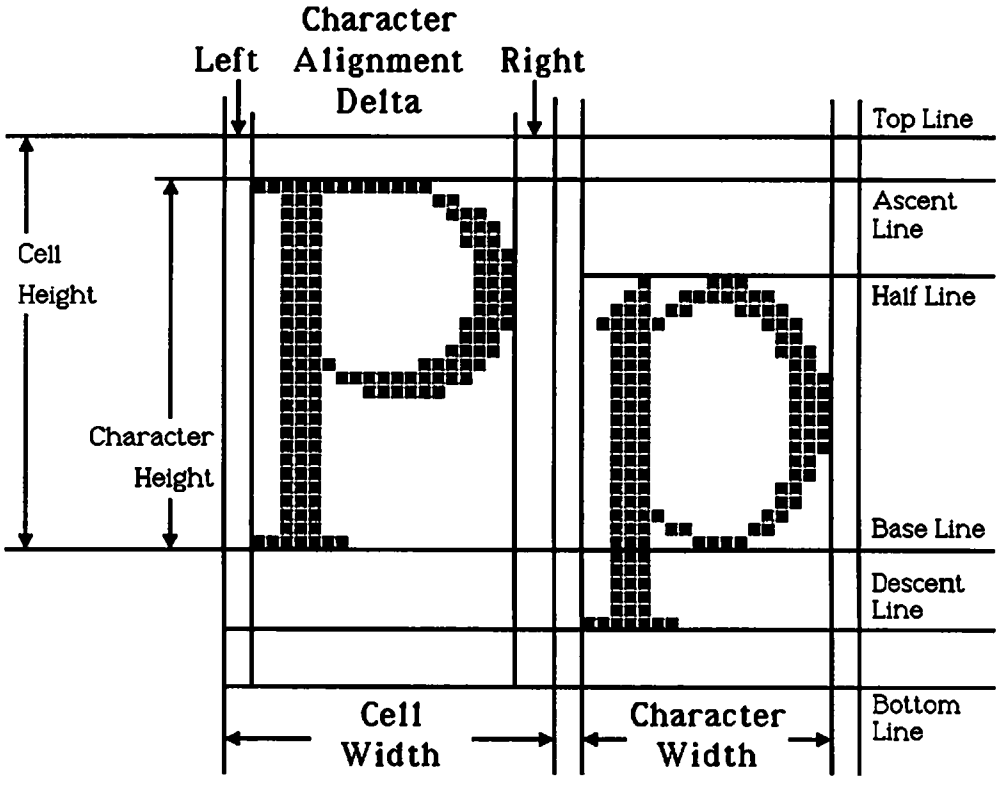


Figure A-1 Character Cell

Draw a filled rectangle with rounded corners. The rectangle is specified by the points in **pxyarray**. All fill area attributes are used.

```
WORD v_rmcur (handle)
        WORD      handle;
```

Remove last graphic cursor. This function removes the last graphic cursor placed on the screen.

```
WORD vro_cpyfm (handle, wr_mode, pxyarray, psrcMFDB,
        pdesMFDB)
        WORD      handle,
                  wr_mode,
                  pxyarray[8];
        FDB       *psrcMFDB, *pdesMFDB;
```

Perform opaque copy raster operation. This function copies a rectangular raster from a source form to a destination form using the writing mode specified. The writing mode is given by **wr_mode** and may be one of the following values:

Constant	*Value*	*Operation*
ALL_WHITE	0	R = 0
S_AND_D	1	R = S AND D
S_AND_NOTD	2	R = S AND (NOT D)
S_ONLY	3	R = S
NOTS_AND_D	4	R = (NOT S) AND D
D_ONLY	5	R = D
S_XOR_D	6	R = S XOR D
S_OR_D	7	R = S OR D
NOT_SORD	8	R = NOT (S OR D)
NOT_SXORD	9	R = NOT (S XOR D)
NOT_D	10	R = NOT D
S_OR_NOTD	11	R = S OR (NOT D)
NOT_S	12	R = NOT S
NOTS_OR_D	13	R = (NOT S) OR D
NOT_SANDD	14	R = NOT (S AND D)
ALL_BLACK	15	R = 1

where S is the source pixel value, D is the destination pixel value, and R is the destination pixel value after the operation. (NOTE: Constant names for modes 10, 11, and 12 may need to be corrected in file OBDEFS.H.)

The **pxyarray** gives the rectangle in the source form to copy from in elements 0 through 3, and the rectangle in the destination to copy to

in elements 4 through 7. If the source and destination rasters are the same, and the source and destination rectangles overlap, the VDI ensures that the source area is copied before it is changed (that is, when it becomes the destination area). The function performs the copy operation pixel for pixel.

If the source and destination rectangles are not the same size, the VDI uses the destination as a pointer and uses the source for the size.

The raster is defined by the FDB structures pointed to by **psrcMFDB** and **pdesMFDB**. The FDB structure is defined in GEMDEFS.H as:

```
typedef struct fdbstr {
    WORD  *fd_addr,        /* raster bit map */
          fd_w,            /* width of raster in pixels */
          fd_h,            /* height of raster in pixels */
          fd_wdwidth,      /* width of raster in words */
          fd_stand,        /* 0=standard, 1=device format */
          fd_nplanes,      /* number of planes */
          fd_r1,           /* reserved */
          fd_r2,
          fd_r3;
} FDB;
```

Refer to Chapter 7 for further details regarding bit maps and rasters. Note that the source and destination rasters must be in the same format specified in **fd_stand**.

```
WORD vrq_choice (handle, ch_in, ch_out)
    WORD    handle,
            ch_in,
            *ch_out;
```

Input choice, request mode. This function waits until input from the choice device is available. The choice key index is returned through **ch_out** as a number from 1 to a device-dependent value. If the key pressed is not a valid choice key, the value in **ch_in** is returned as the selection.

This function is not required and may not be available on all devices.

```
WORD vrq_locator (handle, x, y, xout, yout, term)
    WORD    handle,
            x, y,
            *xout, *yout, *term;
```

Input locator, request mode. This function returns the position of the locator device. When the function is called, it places a cursor at the

initial coordinates given by x and y. The function then tracks the cursor based on the input from the locator. When a terminating event, such as pressing the mouse button or a key, occurs, the cursor is removed and the function returns. The terminating event is indicated in the value of term and the cursor coordinates are returned through xout and yout.

If both the keyboard and another locator device are available, the cursor is tracked by input from either. To determine the type of terminating event, the parameter term is divided into its two bytes. The low byte contains the ASCII value of the character entered, if a keypress was the terminating event. Otherwise, the high byte contains a value, offset by 0x20, indicating the locator button that was pressed. For example, on the mouse, the left button has the value 0x20, the next button to the right is 0x21, etc.

```
WORD vrq_string (handle, max_length, echo_mode, echo_xy, str)
        WORD    handle,
                max_length,
                echo_mode, echo_xy[2];
        char    str[max_length + 1];
```

Input string, request mode. This function accepts characters from the string input device until **max_length** characters have been entered or a carriage return is encountered. If **echo_mode** is TRUE, the characters are echoed to the screen at the coordinates in **echo_xy** using the current text attributes. Otherwise, echoing to the screen is disabled (echoing of input is not required and may not be available on all devices). The characters are returned in **str** as a null-terminated string.

If **max_length** is a negative value, the absolute value is used as above, and the character codes will be the standard keyboard defined in Appendix C. Otherwise, the keycodes may be device-dependent.

```
WORD vrq_valuator (handle, val_in, val_out, term)
        WORD    handle,
                val_in,
                *val_out,
                *term;
```

Input valuator, request mode. This function sets the initial value of the valuator to **val_in**. The final value is returned through **val_out**. The terminating event is returned in **term** (see **vrq_locator( )** for a description of **term**).

In general, the valuator is the up and down arrow keys on the key board. Pressing the up arrow increases the value by ten; pressing the down arrow decreases the value by ten. Pressing these keys with the

shift key increments or decrements the value by one. The range of the valuator is from 1 through 100.

This function is not required and may not be available on all devices.

```
WORD vr_recfl (handle, pxyarray)
        WORD    handle,
                pxyarray[4];
```

Draw a filled rectangular area. The rectangle is given by array **pxyarray.** The current fill area attributes are used, except the perimeter which is not drawn.

```
WORD vrt_cpyfm (handle, wr_mode, pxyarray,
            psrcMFDB, pdesMFDB, color_index)
        WORD    handle,
                wr_mode,
                pxyarray[8];
        FDB     *psrcMFDB, *pdesMFDB;
        WORD    color_index[2];
```

Transparent copy raster operation. This function takes a monochrome raster and copies it to a color raster using the writing mode specified by **wr_mode.** The same writing mode constant names are used for this function as for function **vswr_mode( ).** The effect of each mode is slightly different.

In Replace mode, the destination pixels are replaced. The foreground color index is specified in **color_index[0]** and is output for any source pixel with a value of 1. The background color is specified in **color_index[1]** and is output for any source pixel with a value of 0.

In Transparent mode, the foreground color in **color_index[0]** is output for source pixels with the value of 1. Source pixels with the value 0 have no effect and **color_index[1]** is not used.

In XOR mode, the monochrome raster pixels are logically XORed with *each* plane of the destination raster. The values in **color_index** are not used.

In Reverse Transparent mode, source pixels with the value 0 cause the background color in **color_index[1]** to be output to the destination. Source pixels with the value 1 have no effect, and **color_index[0]** is not used.

The parameters **pxyarray, psrcMFDB,** and **pdesMFDB** are used in the same manner as in function **vro_cpyfm( ).**

```
WORD vr_trnfm (handle, psrcMFDB, pdesMFDB)
        WORD    handle;
        FDB     *pscrMFDB, *pdesMFDB;
```

Transform the format of a raster. This function converts a raster from standard format to device-specific format or vice versa. The raster to transform is defined by the FDB structure pointed to by **psrcMFDB.** The result of the transformation is placed in **pdesMFDB.**

The type of transformation performed is based on the current value of the **fd_stand** field of the FDB structure. If the raster is in standard format, it is converted to device-specific format. If the raster is in device-specific format, it is converted to standard format. In either case, the **fd_stand** field of **pdesMFDB** is set to the resulting format. It is the user's responsibility to set all other fields of the source and destination FDBs.

See Chapter 7 for further discussion about raster and formats. Refer to function **vro_cpyfm( )** for the definition of the FDB structure.

```
WORD v_rvoff (handle)
        WORD    handle;
```

Reverse video off. All subsequent alpha text output is done in normal video.

```
WORD v_rvon (handle)
        WORD    handle;
```

Reverse video on. All subsequent alpha text output is done in reverse video.

```
WORD vsc_form (handle, pcur_form)
        WORD    handle,
                pcur_form[37];
```

Redefine cursor. The cursor is displayed when the **vrq_locator( )** or **v_show_c( )** function is called. The array **pcur_form** has the following structure:

Element	*Description*
0	x coordinate of hot spot
1	y coordinate of hot spot
2	Reserved; must be 1
3	Mask color index (usually 0)
4	Data color index (usually 1)
5–20	16-by-16 bit cursor mask
21–36	16-by-16 bit cursor data

The hot spot is the location of the pixel (relative to the upper left corner of the mouse form) used as the current mouse position.

The mouse form is drawn using the following procedure:

1. The data under the mouse form is saved so that it can be restored when the cursor moves.
2. Bits set to 1 in the mask cause the corresponding pixels to be set to the color index in **pcur_form[3]**.
3. Bits set to 1 in the data cause the corresponding pixels to be set to the color index in **pcur_form[4]**.

```
WORD vs_clip (handle, clip_flag, pxyarray)
        WORD    handle,
                clip_flag,
                pxyarray[4];
```

Set the clipping rectangle for workstation handle. If **clip_flag** is FALSE, clipping by the VDI is turned off. If **clip_flag** is TRUE, clipping is turned on and the clipping rectangle is set to the rectangle in **pxyarray** given in the current coordinate system. The default when a workstation is opened is for clipping to be disabled.

```
WORD vs_color (handle, cindex, rgb_in)
        WORD    handle,
                cindex,
                rgb_in[3];
```

Set the color in the color lookup table. The color index range is device-dependent. On the Atari ST, the indices range from 0 to 15. The color index is given by **cindex**. The color is specified by mixing various intensities of red (**rgb_in[0]**), green (**rgb_in[1]**), and blue (**rgb_in[2]**). The intensity can range from 0, meaning no color, to 1000 for the highest intensity.

If no color lookup table exists, this function performs no operation. On a monochrome device, the VDI maps any percentage of color to white.

```
WORD vs_curaddress (handle, row_num, col_num)
        WORD    handle,
                row_num, col_num;
```

Direct alpha cursor address. This function moves the alpha cursor to the alpha cell specified by coordinates **row_num** and **col_num**. Row and column numbers start at 1. Addresses beyond the displayable range of the screen are set to the nearest value within the displayable area.

```
WORD vsf_color (handle, color_index)
        WORD    handle,
                color_index;
```

Set the color index for subsequent fill polygon operations. The color index specified by **color_index** refers to an index in the color lookup table of the VDI. At least two colors, 0 and 1, are supported. The total number of colors available is device-dependent. If a color index is out of range, the VDI defaults to index 1. The function returns the selected color index.

```
WORD vsf_interior (handle, fstyle)
        WORD    handle,
                fstyle;
```

Set the fill interior style used in subsequent polygon fill operations. The fill style is specified by **fstyle.** Valid style indices are:

0 = hollow
1 = solid
2 = pattern
3 = hatch
4 = user-defined

If an unavailable style index is selected, the VDI defaults to hollow fill. A hollow style fills the interior with the current background color (index 0). Solid style fills the area with the current fill color. Pattern and hatch styles require a further selection through **vsf_style( ).** The user-defined style is set by **vsf_udpat( ).**

The function returns the selected interior-fill style.

```
WORD vsf_perimeter (handle, per_vis)
        WORD    handle,
                per_vis;
```

Set the fill perimeter visibility. When **per_vis** is FALSE, the perimeter visibility is set to off. When **per_vis** is TRUE, it is set to on. Certain polygon drawing operations use the perimeter visibility setting. If the setting is on, a solid perimeter is drawn; otherwise no perimeter is drawn. The function returns the current perimeter visibility setting.

```
WORD vsf_style (handle, fstyle_index)
        WORD    handle,
                fstyle_index;
```

Set the fill style for interior-fill styles of pattern or hatch. If other interior-fill styles are selected, this function has no effect on them. The style index is specified in **fstyle_index**. Valid indices are shown in Figure A-2 and A-3. The total number of patterns and hatches is device-dependent. However, those shown in the figures are always supported. If a requested index is not available, the VDI defaults to index 1. The function returns the index selected.

```
WORD vsf_udpat (handle, pfill_pat, nplanes)
        WORD    handle,
                pfill_pat[16 * nplanes],
                nplanes;
```

Set the user-defined fill pattern. A pattern consists of a 16-by-16 bit array where bits set to 1 indicate the pixel is on. Bit 15 of the first WORD is the first pixel in the pattern. The bit array is pointed to by **pfill_pat**.

The number of planes for multiple-plane patterns is specified by nplanes. Plane 1 of the pattern is held in **pfill_pat** elements 0 through

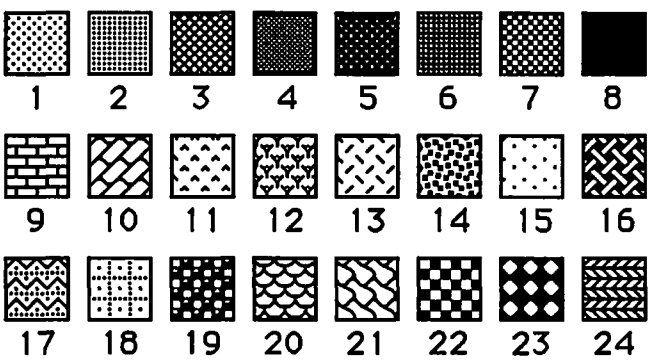


Figure A-2 Fill Patterns

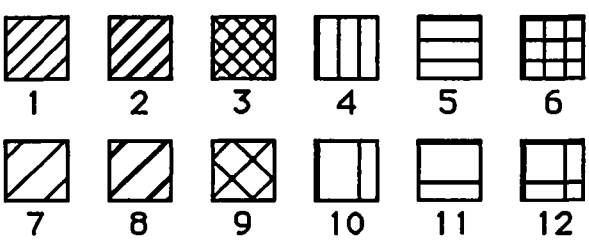


Figure A-3 Fill Hatches

15; plane 2 of the pattern is in elements 16 through 31; etc. Note that the writing mode must be Replace when using multiple-plane patterns.

WORD **v_show_c (handle, reset)**
    WORD **handle,**
    reset;

Show cursor. This function displays the current cursor form. The **v_show_c( )** and **v_hide_c( )** functions are nested. The **v_show_c( )** function must be called as many times as the **v_hide_c( )** function was called since the cursor disappeared for the cursor to become visible again. For example, if the cursor is visible and **v_hide_c( )** is called five times, **v_show_c( )** must be called five times before the cursor reappears. This nesting can be overridden by setting parameter reset to 0. When parameter reset is 0, the number of **v_hide_c( )** calls is ignored and the cursor is made visible. If reset is nonzero, **v_show_c( )** retains its normal functionality.

WORD **vsin_mode (handle, dev_type, in_mode)**
    WORD **handle,**
    dev_type,
    in_mode;

Set the input mode for logical input devices. The input device to set is given by **dev_type.** Valid logical devices are:

1 = locator
2 = valuator
3 = choice
4 = string

The input mode to use is set by **in_mode** where 1 is request mode and 2 is sample mode.

WORD **vsl_color (handle, color_index)**
    WORD **handle,**
    color_index;

Set the color index for subsequent poly-line operations. The color index specified by **color_index** refers to an index in the color lookup table of the VDI. At least two colors, 0 and 1, are supported. The total number of colors available is device-dependent. If a color index is out of range, the VDI defaults to index 1. The function returns the selected color index.

```
WORD vsl_ends (handle, beg_style, end_style)
        WORD    handle,
                beg_style,
                end_style;
```

Select the line end style for subsequent poly-line operations. The start of the line has style **beg_style** and the end of the line has style **end_style.** The style may be any one of the following:

0 = squared (default)
1 = arrow
2 = rounded

The squared and arrow ends stop at the endpoint of the line. The rounded end is drawn so that the center of the rounding is at the endpoint of the line.

```
WORD vsl_type (handle, lstyle)
        WORD    handle,
                lstyle;
```

Set the line type for subsequent poly-line operations. The line style is set by the index given in **lstyle.** The function returns the line style selected. The total number of line styles is device-dependent. All devices support at least six lines styles and one user-defined line style (see Figure A-4).

	Style		16 bits (Bit 15 … Bit 0)
1	_____	solid	1111111111111111
2	_ _ _ _	long dash	1111111111110000
3	- - - - - - - -	dot	1110000011100000
4	_ - - - - - - -	dash, dot	1111111000111000
5	_ _ _ _	dash	1111111100000000
6	- - - - - - - - - - -	dash, dot, dot	1111000110011000
7	_ _ _ _ _	user-defined	1111000000001111

8 - n Device dependent line styles

Figure A-4 Line Styles

The lines are defined by 16 bits, where 1 means the pixel is on and 0 means the pixel is off. If a nondefault line width is used, the device may not thicken the line styles and use only a thickened solid line.

```
WORD vsl_udsty (handle, lpattern)
        WORD    handle,
                lpattern;
```

Set the user-defined line style. When line style index 7 is in use, the line style provided in **lpattern** is used. The pattern consists of 16 bits where the most significant bit is the first pixel in the line.

```
WORD vsl_width (handle, lwidth)
        WORD    handle,
                lwidth;
```

Set the line width for subsequent poly-line operations. Line widths are always odd numbers. The line width requested is given by **lwidth.** If a line width selected is not available, the closest line width not greater than the one requested is chosen. This function is not required and may not be available on all devices. The function returns the line width selected.

```
WORD vsm_choice (handle, ch_out)
        WORD    handle,
                *ch_out;
```

Input choice, sample mode. This function samples the choice input. If input is available, the function returns 1 and the choice selection is returned through **ch_out.** If no input is available, or if an invalid key was pressed, the function returns 0. The choice numbers range from 1 to a device-dependent value.

This function is not required and may not be available on all devices.

```
WORD vsm_color (handle, color_index)
        WORD    handle,
                color_index;
```

Set the color index for subsequent poly-marker operations. The color index specified by **color_index** refers to an index in the color lookup table of the VDI. At least two colors, 0 and 1, are supported. The total number of colors available is device-dependent. If a color index is out of range, the VDI defaults to index 1. The function returns the selected color index.

```
WORD vsm_height (handle, mheight)
        WORD        handle,
                    mheight;
```

Set the height of the markers used in subsequent poly-marker operations. The height is given by **mheight**. If the height for the current marker does not exist, the VDI uses the closest height not larger than the height requested. The function returns the actual height selected.

```
WORD vsm_locator (handle, x, y, xout, yout, term)
        WORD        handle,
                    x, y,
                    *xout, *yout, *term;
```

Input locator, sample mode. This function samples the input from the locator device. No cursor is automatically displayed when this function is called (use the **v_show_c( )** function). The locator starts at the position specified by **x** and **y**. The new coordinates of the locator are returned by **xout** and **yout**. The terminating event, if any, is returned through **term** (see **vrq_locator( )** for a description of **term**). Input is taken from both the keyboard and another locator device if they are available.

The function returns a value of bit 0 set to 1 if the locator coordinates have changed and bit 1 set to 1 if a terminating event had occurred.

```
WORD vsm_string (handle, max_length, echo_mode, echo_xy, str)
        WORD        handle,
                    max_length,
                    echo_mode, echo_xy[2];
        char        str[max_length + 1];
```

Input string, sample mode. This function samples the character input and accepts characters until one of the following events occurs:

data is no longer available,
a carriage return is encountered, or
max_length characters have been read.

If **echo_mode** is TRUE, the characters are echoed to the screen at the coordinates in **echo_xy** using the current text attributes. Otherwise, echoing to the screen is disabled (echoing of input is not required and may not be available on all devices). The characters are returned in **str** as a null-terminated string.

If **max_length** is a negative value, the absolute value is used as above, and the character codes are the standard keyboard as defined in Appendix C. Otherwise, the keycodes may be device-dependent.

The function returns a 0 if no characters were available, or a positive value if characters were available.

```
WORD vsm_type (handle, msymbol)
        WORD    handle,
                msymbol;
```

Select the marker type for subsequent poly-marker operations. The marker selected is given by **msymbol.** The marker index ranges from 1 to a device-dependent value. At least six markers are always defined by the VDI (see Figure A-5). If a marker index is out of range, the VDI defaults to the asterisk (index 3). The function returns the index of the marker selected.

```
WORD vsm_valuator (handle, val_in, val_out, term, status)
        WORD    handle,
                val_in,
                *val_out,
                *term, *status;
```

Input valuator, sample mode. This function tests for any changes to the valuator. If changes have been made, the value of the valuator is altered accordingly. The initial valuator value is given by **val_in.** The resulting value is returned through **val_out.** The terminating event, if any, is returned through **term** (see **vrq_locator**). The status parameter returns 0 if nothing happened, 1 if the valuator has changed, and 2 if a terminating event occurred.

This function is not required and may not be available on all devices.

1	.	Dot
2	+	Plus
3	*	Asterisk
4	□	Square
5	X	Diagonal cross
6	◇	Diamond

7 and up are device dependent

Figure A-5 Poly-marker Types

WORD **vst_alignment (handle, hor_in, vert_in, hor_out, vert_out)**
WORD **handle,**
hor_in, vert_in,
***hor_out, vert_out;**

Set the graphics text alignment for subsequent graphics text operations. Graphics text alignment is referenced on the (x,y) point selected in the output functions **v_gtext( )** and **v_justified( )**. Horizontal alignment has three options:

0 = left justified where the string starts at (x,y)
1 = center justified where the length of the string is centered about (x,y)
2 = right justified where the string ends at (x,y); left justification is the default value.

Vertical alignment has 6 options:

0 = baseline (default)
1 = half line
2 = ascent line
3 = bottom line
4 = descent line
5 = top line

For vertical alignment, the line chosen passes through the point (x,y) (see Figure A-6 for line placement).

Horizontal and vertical alignment are selected separately by setting **hor_in** and **vert_in**, respectively. The actual alignment values being used by the VDI are returned through **hor_out** and **vert_out**. If an invalid alignment value is requested, the VDI chooses the default value.

WORD **vst_color (handle, color_index)**
WORD **handle,**
color_index;

Set the color index for subsequent graphic text operations. The color index specified by **color_index** refers to an index in the color lookup table of the VDI. At least two colors, 0 and 1, are supported. The total number of colors available is device-dependent. If a color index is out of range, the VDI defaults to index 1. The function returns the selected color index.

WORD **vst_effects (handle, teffect)**
WORD **handle,**
teffect;

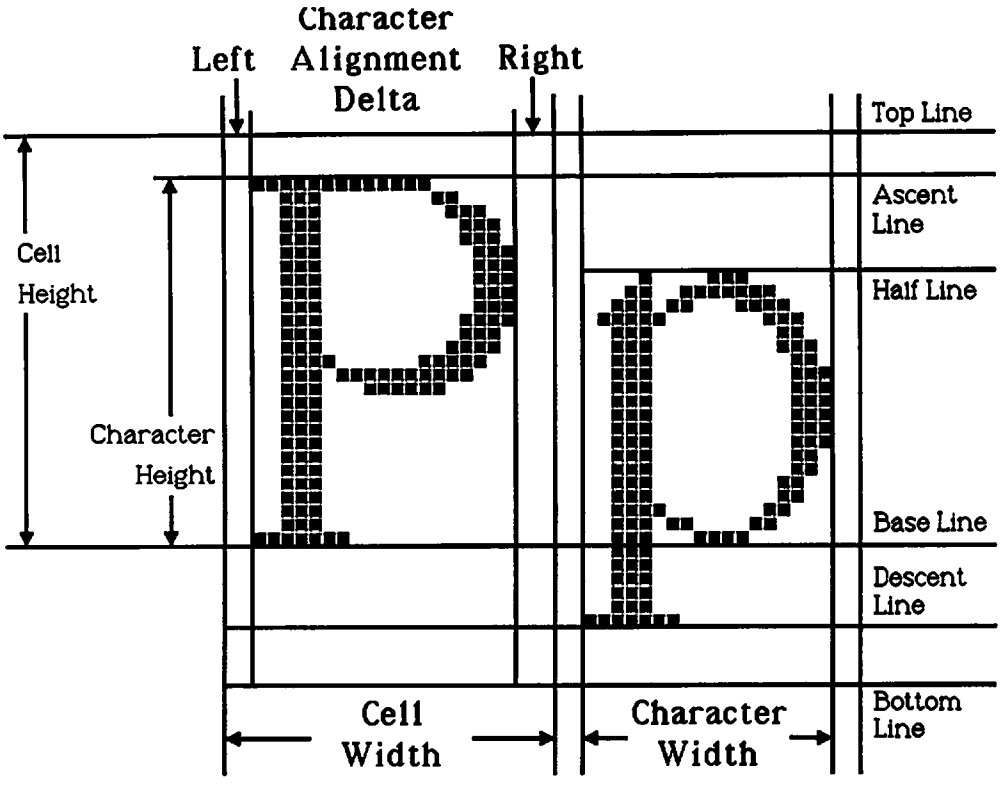


Figure A-6 Character Cell

Set the text special effects for subsequent graphic text operations. The bit settings of **teffect** determine which effects are active. The bit descriptions are:

Bit	*Description*
0	Thicken
1	Light intensity
2	Skew (like italic)
3	Underline
4	Outline
5	Shadow

When the bit is set, the text output has that attribute. Any combination of text effects is allowed.

The function returns the effect setting selected.

```
WORD vst_font (handle, tfont)
        WORD    handle;
                tfont;
```

Select the graphic character face for subsequent graphic text operations. The face index to use is specified by **tfont.** Some face names and indices are:

Index	*Name*
1	System face
2	Swiss 721
3	Swiss 721 Thin
4	Swiss 721 Thin Italic
5	Swiss 721 Light
6	Swiss 721 Light Italic
7	Swiss 721 Italic
8	Swiss 721 Bold
9	Swiss 721 Bold Italic
10	Swiss 721 Heavy
11	Swiss 721 Heavy Italic
12	Swiss 721 Black
13	Swiss 721 Black Italic
14	Dutch 801 Roman
15	Dutch 801 Italic
16	Dutch 801 Bold
17	Dutch 801 Bold Italic

Only face 1 is built into the VDI. All other faces must be loaded from face files using the **vst_load_fonts( )** function.

The function returns the index of the font selected.

```
WORD vst_height (handle, theight, char_width, char_height,
            cell_width, cell_height)
        WORD    handle,
                theight,
                *char_width, *char_height,
                *cell_width, *cell_height;
```

Set the current text height in coordinate units. The height, specified in **theight,** is the distance from the character baseline to the top of the character cell (see Figure A-7).

The function sets **char_width** and **char_height** to the character width and height, respectively. The cell width and cell height are

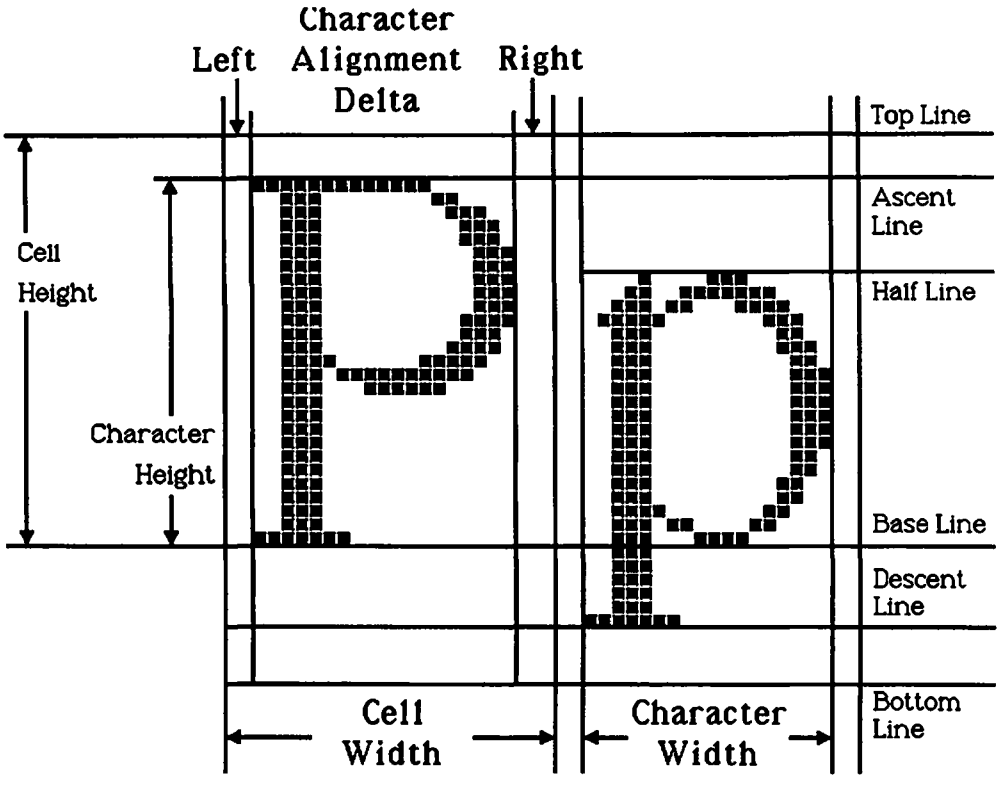


Figure A-7 Character Cell

placed into **cell_width** and **cell_height**, respectively. For proportionally spaced fonts, the VDI returns the width of the widest character.

If the requested size does not exist, the VDI uses the closest character size that does not exceed the requested size.

WORD **vst_load_fonts (handle, select)**
WORD **handle, select;**

Load the fonts associated with a particular driver. The fonts are linked to the driver through the ASSIGN.SYS file. Once the file is loaded, the fonts are available to the workstation specified by handle.

The function returns the number of newly generated font identifiers. If the fonts were already available to the workstation, no action occurs and the function returns 0. The select parameter is reserved and should always be 0.

WORD **vst_point (handle, tpoint, char_width, char_height, cell_width, cell_height)**

```
WORD    handle,
        tpoint,
        *char_width, *char_height,
        *cell_width, *cell_height;
```

Set the current text height in points. One point is 1/72 of an inch. The point size, specified in **tpoint**, is the distance from the baseline of one line of text to the baseline of the next line of text (see Figure A-8).

The function sets **char_width** and **char_height** to the character width and height, respectively. The cell width and cell height are placed into **cell_width** and **cell_height**, respectively. For proportionally spaced fonts, the VDI returns the width of the widest character. These values are given in current coordinate units.

If the requested size does not exist, the VDI uses the closest character size that does not exceed the requested size.

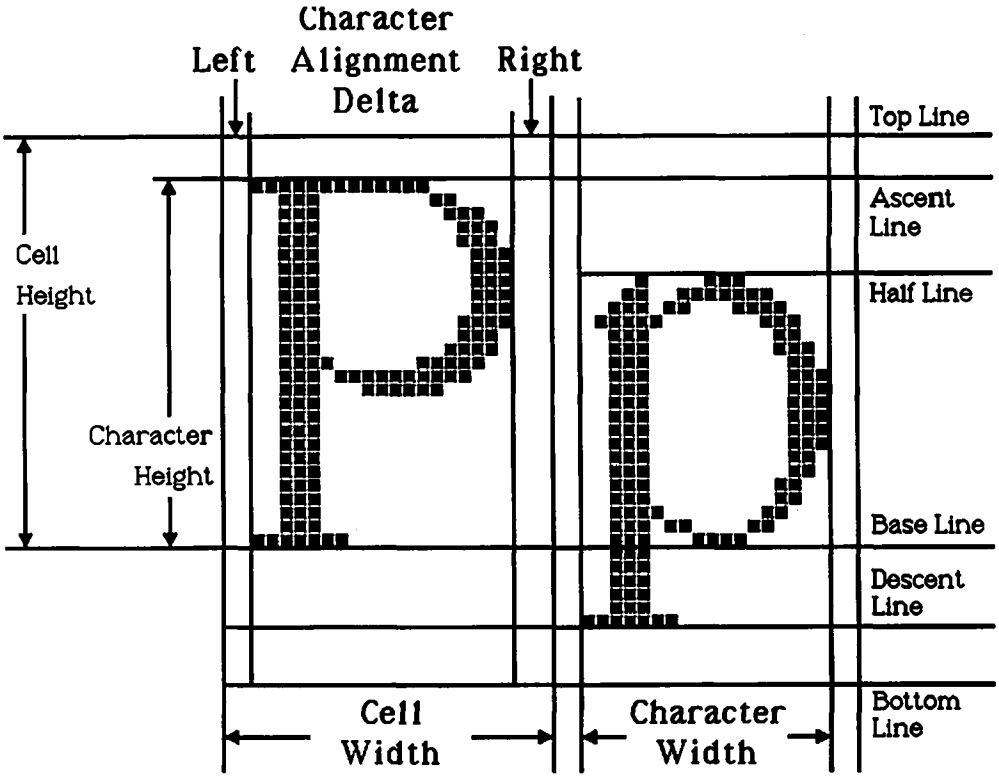


Figure A-8 Character Cell

WORD **vst_rotation (handle, txtangle)**
WORD **handle,**
txtangle;

Set the angle of rotation of the character baseline. The angle of rotation, given in **txtangle**, measures tenths of a degree and ranges from 0 to 3600. The angle 0 is the default text rotation and increasing angles continue counterclockwise.

The baseline angle used is a best-fit value. Some devices do not support text rotation, and other devices allow only quarter-turn rotations (0, 90, and 270 degrees). The function returns the actual angle of rotation selected.

WORD **vst_unload_fonts (handle, select)**
WORD **handle, select;**

Remove the fonts from a workstation. This function makes the fonts loaded through the **vst_load_fonts( )** function unavailable to the workstation given by **handle.** If the fonts are being used by more than one workstation with the same physical device handle, the fonts are not unloaded from memory until one of two conditions exists. Either all workstations that share the fonts are closed, or all workstations that share the fonts request that the fonts be unloaded. The default system fonts for the workstation always remain loaded and available. The parameter **select** is reserved and should always be 0.

WORD **vswr_mode (handle, wmode)**
WORD **handle,**
wmode;

Set the writing mode for the workstation to the value of **wmode.** Valid writing modes are:

Constant	*Value*	*Mode*
MD_REPLACE	1	Replace
MD_TRANS	2	Transparent
MD_XOR	3	XOR
MD_ERASE	4	Reverse transparent

The resulting output on the display is determined by the output pixel value (source) and the current pixel value (destination). The following definitions are used:

Term	*Definition*
Mask	Output pixel value of the image
Fore	Selected output color pixel chosen
Back	Background color pixel in use (color 0)
Old	Current pixel value
New	Resulting pixel value

Replace mode simply outputs the image regardless of the current contents of the display:

new = (fore AND mask) OR (back AND NOT mask)

Transparent mode affects those pixels where the image is to appear:

new = (fore AND mask) OR (old AND NOT mask)

XOR mode reverses the bits representing the color:

new = mask XOR old

Reverse Transparent mode affects those pixels where the image does not appear:

new = (old AND mask) OR (fore AND NOT mask)

The function returns the writing mode selected. The writing mode stays in effect until it is explicitly changed. The default writing mode is Replace.

VOID **Vsync ()**

Halt program execution until the next vertical blank interrupt. This function is useful for synchronizing graphic output.

WORD **v_updwk (handle)**
    WORD **handle;**

Update the workstation. This function causes all pending graphic commands for the workstation given by **handle** to be executed immediately. The commands are executed in the order in which they were placed in the buffer. For printer or plotter devices, this function causes the device drive to begin output to the device. For a metafile, GEM VDI outputs the opcode to update the workstation. This function has no effect on screen devices.

```
WORD wind_calc (wi_ctype, wi_ckind,
            wi_cinx, wi_ciny, wi_cinw, wi_cinh,
            wi_coutx, wi_couty, wi_coutw, wi_couth)
      WORD      wi_ctype, wi_ckind,
                wi_cinx, wi_ciny, wi_cinw, wi_cinh,
                *wi_coutx, *wi_couty, *wi_coutw, *wi_couth;
```

Calculate the size and position of a window's border area or work area. When given the work area, the function calculates the border area. When given the border area, the function calculates the work area. The type of information returned is determined by **wi_ctype**, where:

0 = return border area data
1 = return work area data

The bits of parameter **wi_ckind** indicate the window control areas to include in the calculation. See **wi_crkind** of function **wind_create( )** for the bit settings.

The input area is passed through parameters **wi_cinx**, **wi_ciny**, **wi_cinw**, and **wi_cinh**. The calculated area is returned through parameters **wi_coutx**, **wi_couty**, **wi_coutw**, and **wi_couth**.

A returned value of 0 indicates an error.

```
WORD wind_close (wi_clhandle)
      WORD      wi_clhandle;
```

Close a window to remove it from the screen. The window specified by **wi_clhandle** is removed from the screen. This function only removes the window from visibility. The window is still kept in the AES. A returned value of 0 indicates an error.

```
WORD wind_create (wi_crkind, wi_crwx, wi_crwy, wi_crww,
            wi_crwh)
      WORD      wi_crkind,
                wi_crwx, wi_crwy, wi_crww, wi_crwh;
```

Allocate the application's full-size window and return the window handle. This function requests allocation of a window from the AES. GEM on the Atari ST allows up to eight windows to be allocated at any given time. The bits of parameter **wi_crkind** determine the window components to be displayed as follows:

Constant	*Value*	*Component*
NAME	0x0001	Title bar with name
CLOSER	0x0002	Close box
FULLER	0x0004	Full box
MOVER	0x0008	Move box
INFO	0x0010	Information line
SIZER	0x0020	Size box
UPARROW	0x0040	Up arrow
DNARROW	0x0080	Down arrow
VSLIDE	0x0100	Vertical slider
LFARROW	0x0200	Left arrow
RTARROW	0x0400	Right arrow
HSLIDE	0x0800	Horizontal slider

If the bit is set to 1, the window displays that component. For a description of the components, see the text.

The parameters **wi_crwx**, **wi_crwy**, **wi_crww**, and **wi_crwh** define the window's position and size when at its largest size.

The returned value ranges from 0 to an environment-specific value (8 on the ST). This value is the window handle used by the other Window Manager functions. If a negative value is returned, the GEM AES has no more windows available.

```
WORD wind_delete (wi_dhandle)
        WORD      wi_dhandle;
```

Delete a window from the AES. When an application no longer requires the use of a window (e.g., prior to exiting to the Desktop), the application should release the allocated portion of the AES being used to maintain the window. The window should also be closed before it is deleted. A returned value of 0 indicates an error.

```
WORD wind_find (wi_fmx, wi_fmy)
        WORD      wi_fmx, wi_fmy;
```

Get the handle of the first window under the location given. This function locates the window being displayed at the location specified by **wi_fmx** and **wi_fmy** and returns the handle to the window found, or 0 if it is the Desktop.

```
WORD wind_get (wi_ghandle, wi_gfield,
               wi_gw1, wi_gw2, wi_gw3, wi_gw4)
        WORD      wi_ghandle, wi_gfield,
                  *wi_gw1, *wi_gw2, *wi_gw3, *wi_gw4;
```

Obtain information regarding a window. This is a multi-purpose function. The window handle is given by **wi_ghandle.** The information is returned through parameters **wi_gw1, wi_gw2, wi_gw3,** and **wi_gw4.** The type of information to return is specified with **wi_gfield.** The predefined constant names and values and the usage of the return parameter are given below:

WF_WORKXYWH 4
: request the size and position of the window's work area
  - **wi_gw1**: x coordinate of upper left corner
  - **wi_gw2**: y coordinate of upper left corner
  - **wi_gw3**: width of work area
  - **wi_gw4**: height of work area

WF_CURRXYWH 5
: request the size and position of the entire current window including control areas
  - **wi_gw1**: x coordinate of upper left corner
  - **wi_gw2**: y coordinate of upper left corner
  - **wi_gw3**: width of current window
  - **wi_gw4**: height of current window

WF_PREVXYWH 6
: request the window's previous size and position
  - **wi_gw1**: x coordinate of upper left corner
  - **wi_gw2**: y coordinate of upper left corner
  - **wi_gw3**: width of previous window
  - **wi_gw4**: height of previous window

WF_FULLXYWH 7
: request the full window's size and position
  - **wi_gw1**: x coordinate of upper left corner
  - **wi_gw2**: y coordinate of upper left corner
  - **wi_gw3**: width of full window
  - **wi_gw4**: height of full window

WF_HSLIDE 8
: request relative position of horizontal slider
  - **wi_gw1**: value from 0 (furthest left) to 100 (furthest right)

WF_VSLIDE 9
: request relative position of vertical slider
  - **wi_gw1**: value from 0 (top) to 1000 (bottom)

WF_TOP 10
: request the handle of the active (topmost) window
  - **wi_gw1**: handle of active window

WF_FIRSTXYWH 11
: request position and size of the first rectangle in the window's visible rectangle list (see text)

wi_gw1: x coordinate of upper left corner
wi_gw2: y coordinate of upper left corner
wi_gw3: width
wi_gw4: height

WF_NEXTXYWH 12
request position and size of the next rectangle in the window's visible rectangle list (see text)
wi_gw1: x coordinate of upper left corner
wi_gw2: y coordinate of upper left corner
wi_gw3: width
wi_gw4: height

WF_HSLSIZE 15
request size of the horizontal slider
wi_gw1: 1 to 1000 = size *relative* to scroll bar
−1 = default minimum size (square box)

WF_VSLSIZE 16
request size of the vertical slider
wi_gw1: 1 to 1000 = size *relative* to scroll her
−1 = default minimum size (square box)

WF_SCREEN 17
request address and length of the internal menu/alert buffers
wi_gw1: low WORD of address
wi_gw2: high WORD of address
wi_gw3: low WORD of length
wi_gw4: high WORD of length

A returned value of 0 indicates an error.

WORD **wind_open (wi_ohandle, wi_owx, wi_owy, wi_oww, wi_owh)**
WORD **wi_ohandle,**
wi_owx, wi_owy, wi_oww, wi_owh;

Open a window to make it visible on the screen. The window to open is specified by its handle given in **wi_ohandle.** The initial position and size of the window are given by **wi_owx, wi_owy, wi_oww,** and **wi_owh.** A returned value of 0 indicates an error.

WORD **wind_set (wi_shandle, wi_sfield,**
wi_sw1, wi_sw2, wi_sw3, wi_sw4)
WORD **wi_shandle, wi_sfield,**
wi_sw1, wi_sw2, wi_sw3, wi_sw4;

Change the values in a window's field that determine how the window is displayed. The window is specified by the handle in **wi_shandle.** The

field to change is given by **wi_sfield** using the predefined constants given below. The remaining parameters hold the new values for the field to change. These parameters are defined as WORDs but may also be used to pass address pointers. Because an address requires two WORDs, a program may simply use the address in the parameter list in place of two WORD parameters (either **wi_sw1** and **wi_sw2** or **wi_sw3** and **wi_sw4**). For example, to set the information line, the call

wind_set (wi_shandle, WF_INFO, info_line);

could be used where parameter **info_line** is a pointer to a string. The predefined constants and parameter usage are:

WF_KIND 1
set the components of the window
wi_sw1: see wi_crkind in the wind_create() function for the appropriate bit settings

WF_NAME 2
set the name in the title bar
wi_sw1 and wi_sw2: pointer to a string

WF_INFO 3
set the information line
wi_sw1 and wi_sw2: pointer to a string

WF_CURRXYWH 5
set the size and position of the entire current window including control areas
wi_sw1: x coordinate of upper left corner
wi_sw2: y coordinate of upper left corner
wi_sw3: width of current window
wi_sw4: height of current window

WF_HSLIDE 8
set the relative position of horizontal slider
wi_sw1: value from 0 (furthest left) to 1000 (furthest right)

WF_VSLIDE 9
set the relative position of vertical slider
wi_sw1: value from 0 (top) to 1000 (bottom)

WF_TOP 10
set a new active (topmost) window
wi_sw1: handle of active window

WF_FIRSTXYWH 14
set the address of a new default GEM Desktop for the GEM AES to draw
wi_sw1 and wi_sw2: address of the object tree
wi_sw3: index of starting object to draw

WF_HSLSIZE 15
set the size of the horizontal slider
wi_sw1: 1 to 1000 = size *relative* to scroll bar
−1 = default minimum size (square box)
WF_VSLSIZE 16
set the size of the vertical slider
wi_sw1: 1 to 1000 = size *relative* to scroll bar
−1 = default minimum size (square box)

A returned value of 0 indicates an error.

WORD **wind_update (wi_ubegend)**
WORD **wi_ubegend;**

Communicate with the AES about the application's current processing. This function informs the AES of four specific conditions. The condition is identified by the pre-defined constant given in **wi_ubegend.** The constants and conditions are defined:

END_UPDATE 0
Notify the AES that the application has completed updating a window. The AES again allows the user to interact with the screen.
BEG_UPDATE 1
Notify the AES that the application is about to update a window. During the update, the AES does not allow the user to interact with the windows.
END_MCTRL 2
Notify the AES that the application is returning control of all mouse functions to the AES.
BEG_MCTRL 3
Notify the AES that the application is taking control of all mouse functions. While the application has control of the mouse functions, the Screen Manager does not respond to mouse input, and menus and window control are not active unless handled by the application.

The function returns 0 if an error condition exists.

VOID **Xbtimer (timerset, ctrlreg, data, vec)**
WORD **timerset, ctrlreg, data;**
WORD **(*vec) ();**

Set the timers on the 68901 MFP (multi-function peripheral). The timer to set is given by **timerset.** An 8-bit value for the timer control

register is given by **ctrlreg.** The 8 bit data value for the timer data register is given by **data.** The new interrupt vector is given by **vec.** Refer to the 68901 for details of its operation.

GEM DOS Functions

Cauxin	Read character from AUX:
Cauxis	Check status of AUX: input
Cauxos	Check status of AUX: output
Cauxout	Write character to AUX:
Cconin	Read character standard input
Cconis	Check status of standard input
Cconos	Check status of standard output
Cconout	Write character to standard output
Cconrs	Read edited string from standard input
Cconws	Write string to standard input
Cnecin	Read character from standard input, no echo
Cprnos	Check status of PRN:
Cprnout	Write character to PRN:
Crawcin	Raw input from standard input
Crawio	Raw I/O with standard input/output
Dcreate	Create directory
Ddelete	Delete directory
Dfree	Get drive free space
Dgerdrv	Get default drive
Dgetpath	Get current directory
Dstdrv	Set default drive
Dsetpath	Set current path
Fattrib	Get or set file attributes
Fclose	Close file
Fcreate	Create file
Fdatime	Get or set file time stamp
Fdelete	Delete file
Fdup	Duplicate file handle
Fforce	Force file handle
Fgetdta	Get disk transfer address (DTA)
Fopen	Open file
Fread	Read from file
Frename	Rename file
Fseek	Seek file pointer
Fsetdta	Set DTA
Fsfirst	Search for first file
Fsnext	Search for next file
Fwrite	Write to file
Malloc	Allocate memory

Mfree	Release memory
Mshrink	Shrink size of allocated block
Pexec	Load and execute process
Pterm	Terminate process with exit code
Pterm0	Terminate process with exit code 0
Ptermres	Terminate and stay resident
Super	Get, set, or inquire supervisor mode
Sversion	Get GEM version number
Tgetdate	Get date
Tgettime	Get time
Tsetdate	Set date
Tsettime	Set time

GEM BIOS Functions

Bconin	Input from character device
Bconout	Output to character device
Bconstat	Get character device input status
Bcostat	Get status of output device
Drvmap	Get map of active drives
Getbpb	Get BIOS parameter block
Getmpb	Get memory parameter block
Kbshift	Get or set keyboard shift bits
Mediach	Check for media change
Rwabs	Read or write block on a device
Setexc	Set exception vector
Tickcal	Get system timer calibration

Atari ST Extended BIOS (XBIOS) Functions

Bioskeys	Restore keyboard translation tables
Cursconf	Configure terminal cursor
Dosound	Set sound daemon program counter
Flopfmt	Format track on floppy drive
Floprd	Read from floppy drive
Flopver	Verify sectors on floppy drive
Flopwr	Write to floppy drive
Getrez	Get screen's current resolution
Gettime	Get time and date
Giaccess	Read or write registers on sound chip
Ikbdws	Write string to intelligent keyboard
Initmous	Initialize mouse packet handler
Iorec	Get address of serial device buffer records
Jdisint	Disable interrupt on MFP
Jenabint	Enable interrupt on MFP
Kbrate	Get or set keyboard's repeat rate

Kbdvbase	Get keyboard device packet vectors
Keytbl	Set keyboard translation tables
Logbase	Get screen's logical base address
Mfpint	Set MFP interrupt number
Midiws	Write string to MIDI port
Offgibit	Clear Port A bit on sound chip
Ongibit	Set Port A bit on sound chip
Physbase	Get screen's physical base address
Protobt	Prototype a boot sector
Puntaes	Throw away AES
Random	Generate a random number
Rsconf	Configure RS232 port
Scrdmp	Dump screen to printer
Setcolor	Set color register in palette
Setpalette	Set hardware color palette
Setprt	Set or get printer configuration byte
Setscreen	Set screen addresses and resolution
Settime	Set time and date
Supexec	Execute code in supervisor mode
Vsync	Wait for vertical blank interrupt
Xbtimer	Set MFP timer control

AES Application Manager Functions

appl_exit	Exit application from AES
appl_find	Locate the ID of another application
appl_init	Initialize application to AES
appl_read	Read from the message pipe
appl_tplay	Play GEM AES recording of user's actions
appl_trecord	Record user's actions
appl_write	Write to the message pipe

AES Event Manager Functions

evnt_button	Wait for mouse button event
evnt_dclick	Get or set double-click speed
evnt_keybd	Wait for keyboard event
evnt_mesag	Wait for message event
evnt_mouse	Wait for mouse event
evnt_multi	Wait for multiple events
evnt_timer	Wait for timer event

AES File Selector Manager Function

fsel_input	Display file selector dialog box

AES Form Manager Functions

form_alert	Display alert box
form_center	Center dialog box on screen
form_dial	Prepare screen for dialog box
form_do	Monitor user interactions with form
form_error	Display error box

AES Graphics Manager Functions

graf_dragbox	Draw and track moving box
graf_growbox	Draw an expanding box
graf_handle	Get VDI handle for screen workstation
graf_mbox	Draw a moving box
graf_mkstate	Return current mouse state
graf_mouse	Change mouse form
graf_rubberbox	Draw and track rubber box
graf_shrinkbox	Draw a shrinking box
graf_slidebox	Track sliding box in its parent
graf_watchbox	Track a watch box

AES Menu Manager Functions

menu_bar	Display or erase the menu bar
menu_icheck	Display or erase check mark in menu item
menu_ienable	Display menu item as enabled or disabled
menu_register	Register desk accessory with AES
menu_text	Change text of menu item
menu_tnormal	Display menu title in reverse or normal video

AES Object Manager

objc_add	Add object to an object tree
objc_change	Change an object's state
objc_delete	Delete an object from an object tree
objc_draw	Draw an object or object tree
objc_edit	Let user edit text object
objc_find	Locate object under mouse
objc_offset	Compute object's location relative to the screen
objc_order	Change the order of an object within the tree

AES Resource Manager

rsrc_free	Remove resource file from memory
rsrc_gaddr	Get the address of a data structure in memory

rsrc_load	Load a resource file into memory
rsrc_obfix	Convert object's x and y coordinates
rsrc_saddr	Store the address of a data structure

AES Scrap Manager

scrp_read	Read the scrap directory in the clipboard
scrp_write	Write the scrap directory to the clipboard

AES Shell Manager

shel_envron	Search environment for parameter
shel_find	Locate a filename
shel_read	Identify the command that invoked the application
shel_write	Exit AES or run another application

AES Window Manager

wind_calc	Calculate window area
wind_close	Close an open window
wind_create	Allocate window and obtain handle
wind_delete	Remove window from AES
wind_find	Locate window under given coordinates
wind_get	Get window values
wind_open	Open a created window
wind_set	Set window values
wind_update	Notify AES about window updating

VDI Control Functions

v_clrwk	Clear workstation
v_clsvwk	Close virtual workstation
v_clswk	Close workstation
v_opnvwk	Open virtual workstation
v_opnwk	Open workstation
v_updwk	Update workstation
vst_load_fonts	Load fonts for device
vst_unload_fonts	Remove fonts for device
vs_clip	Set clipping rectangle

VDI Output Functions

v_arc	Arc
v_bar	Bar

v_circle	Circle
v_cellarray	Cell array
v_contourfill	Contour fill
v_ellarc	Elliptical arc
v_ellipse	Ellipse
v_ellpie	Elliptical pie
v_fillarea	Fill area
v_gtext	Graphics text
v_justified	Justified graphics text
v_pieslice	Pie
v_pline	Poly-lines
v_pmarker	Poly-markers
v_rbox	Rounded rectangle
v_rfbox	Filled rounded rectangle
vr_recfl	Filled rectangle

VDI Attribute Functions

vs_color	Set color representation
vsf_color	Set fill color
vsf_interior	Set interior-fill style
vsf_perimeter	Set fill perimeter visibility
vsf_style	Set fill style index
vsf_udpat	Set user-defined fill pattern
vsl_color	Set poly-line color index
vsl_ends	Set poly-line end styles
vsl_type	Set poly-line line type
vsl_udsty	Set user-defined line style pattern
vsl_width	Set poly-line line width
vsm_color	Set poly-marker color index
vsm_height	Set poly-marker height
vsm_type	Set poly-marker type
vst_alignment	Set graphics text alignment
vst_color	Set graphics text color index
vst_effects	Set graphics text effects
vst_font	Set text face
vst_height	Set character height, absolute
vst_point	Set character height, points
vst_rotation	Set character baseline rotation
vswr_mode	Set writing mode

VDI Raster Functions

v_get_pixel	Get pixel
vro_cpyfm	Copy raster, opaque

vrt_cpyfm	Copy raster, transparent
vr_trnfm	Transform form

VDI Input Functions

vex_butv	Exchange button change vector
vex_curv	Exchange cursor change vector
vex_motv	Exchange mouse movement vector
vex_timv	Exchange timer interrupt vector
v_hide_c	Hide cursor
vq_mouse	Sample mouse button state
vq_key_s	Sample keyboard state information
vrq_choice	Input choice, request mode
vrq_locator	Input locator, request mode
vrq_string	Input string, request mode
vrq_valuator	Input valuator, request mode
vsc_form	Set mouse form
v_show_c	Show cursor
vsin_mode	Set input mode
vsm_choice	Input choice, sample mode
vsm_locator	Input locator, sample mode
vsm_string	Input string, sample mode
vsm_valuator	Input valuator, sample mode

VDI Inquire Functions

vq_cellarray	Inquire cell array
vq_color	Inquire color representation
vq_extnd	Extended inquire
vqf_attributes	Inquire fill area attributes
vqin_mode	Inquire input mode
vql_attributes	Inquire poly-line attributes
vqm_attributes	Inquire poly-marker attributes
vqt_attributes	Inquire current graphics text attributes
vqt_extent	Inquire text extent
vqt_fontinfo	Inquire current face information
vqt_name	Inquire face name and index
vqt_width	Inquire character cell width

VDI Escape Functions

V_bit_image	Output bit image file
v_clear_disp_list	Clear display list
v_curdown	Alpha cursor down

v_curhome	Home Alpha cursor
v_curleft	Alpha cursor left
v_curright	Alpha cursor right
v_curtext	Output cursor addressable alpha text
v_curup	Alpha cursor up
v_dspcur	Place graphics cursor at location
v_eeol	Erase to end of alpha text line
v_eeos	Erase to end of alpha screen
v_enter_cur	Exit alpha mode
v_exit_cur	Enter alpha mode
v_form_adv	Form advance
v_hardcopy	Hard copy
v_meta_extents	Update metafile extents
vm_filename	Change GEM VDI file name
v_output window	Output window
vq_chcells	Inquire addressable alpha character cells
vq_curaddress	Inquire current alpha cursor address
vq_tabstatus	Inquire tablet status
v_rmcur	Remove last graphics cursor
v_rvoff	Reverse video off
v_rvon	Reverse video on
vs_curaddress	Direct alpha cursor address

APPENDIX B

Header Files

This appendix contains the source code listings for the header files mentioned in this book. These files are used by the Megamax C compiler and the Atari development system. Depending upon the compiler and the version of GEM you are using, you may find a few changes to these header files. When using the programs in this book, if you get compiler errors indicating undefined constants or function names, you should compare the header files you are using with the ones listed here. Then make the appropriate changes to your header files or the program files.

GEMBIND.H

```
/*********************************************************************/
/*      GEMBIND.H Do-It-Yourself GEM binding kit.                    */
/*            Copyright 1985 Atari Corp.                             */
/*                                                                   */
/*      WARNING: This file is not supported!                         */
/*               We reccomend you use the supplied binding libraries */
/*********************************************************************/
                                  /* Application Manager             */
#define APPL_INIT 10
#define APPL_READ 11
#define APPL_WRITE 12
#define APPL_FIND 13
#define APPL_TPLAY 14
#define APPL_TRECORD 15
#define APPL_EXIT 19
                                  /* Event Manager                   */
#define EVNT_KEYBD 20
#define EVNT_BUTTON 21
#define EVNT_MOUSE 22
#define EVNT_MESAG 23
```

```
#define EVNT_TIMER 24
#define EVNT_MULTI 25
#define EVNT_DCLICK 26
                                /* Menu Manager                    */
#define MENU_BAR 30
#define MENU_ICHECK 31
#define MENU_IENABLE 32
#define MENU_TNORMAL 33
#define MENU_TEXT 34
#define MENU_REGISTER 35
                                /* Object Manager                  */
#define OBJC_ADD 40
#define OBJC_DELETE 41
#define OBJC_DRAW 42
#define OBJC_FIND 43
#define OBJC_OFFSET 44
#define OBJC_ORDER 45
#define OBJC_EDIT 46
#define OBJC_CHANGE 47
                                /* Form Manager                    */
#define FORM_DO 50
#define FORM_DIAL 51
#define FORM_ALERT 52
#define FORM_ERROR 53
#define FORM_CENTER 54
#define FORM_KEYBD 55
#define FORM_BUTTON 56
                                /* Graphics Manager                */
#define GRAF_RUBBOX 70
#define GRAF_DRAGBOX 71
#define GRAF_MBOX 72
#define GRAF_GROWBOX 73
#define GRAF_SHRINKBOX 74
#define GRAF_WATCHBOX 75
#define GRAF_SLIDEBOX 76
#define GRAF_HANDLE 77
#define GRAF_MOUSE 78
#define GRAF_MKSTATE 79
                                /* Scrap Manager                   */
#define SCRP_READ 80
#define SCRP_WRITE 81
                                /* File Selector Manager           */
#define FSEL_INPUT 90
                                /* Window Manager                  */
#define WIND_CREATE 100
#define WIND_OPEN 101
#define WIND_CLOSE 102
#define WIND_DELETE 103
#define WIND_GET 104
#define WIND_SET 105
#define WIND_FIND 106
#define WIND_UPDATE 107
#define WIND_CALC 108
                                /* Resource Manager                */
```

```
#define RSRC_LOAD 110
#define RSRC_FREE 111
#define RSRC_GADDR 112
#define RSRC_SADDR 113
#define RSRC_OBFIX 114
                                        /* Shell Manager                    */
#define SHEL_READ 120
#define SHEL_WRITE 121
#define SHEL_GET 122
#define SHEL_PUT 123
#define SHEL_FIND 124
#define SHEL_ENVRN 125
                                            /* max sizes for arrays         */
#define C_SIZE 4
#define G_SIZE 15
#define I_SIZE 16
#define O_SIZE 7
#define AI_SIZE 2
#define AO_SIZE 1
                                            /* Crystal funtion op code      */
#define OP_CODE control[0]
#define IN_LEN control[1]
#define OUT_LEN control[2]
#define AIN_LEN control[3]

#define RET_CODE int_out[0]
                                            /* application lib parameters   */
#define AP_VERSION global[0]
#define AP_COUNT global[1]
#define AP_ID global[2]
#define AP_LOPRIVATE global[3]
#define AP_HIPRIVATE global[4]
#define AP_LOPNAME global[5]                /* long ptr. to tree base in rsc*/
#define AP_HIPNAME global[6]
#define AP_LO1RESV global[7]                /* long address of memory alloc.*/
#define AP_HI1RESV global[0]
#define AP_LO2RESV global[9]                /* length of memory allocated   */
#define AP_HI2RESV global[10]               /* colors available on screen   */
#define AP_LO3RESV global[11]
#define AP_HI3RESV global[12]
#define AP_LO4RESV global[13]
#define AP_HI4RESV global[14]

#define AP_GLSIZE int_out[1]

#define AP_RWID int_in[0]
#define AP_LENGTH int_in[1]
#define AP_PBUFF addr_in[0]

#define AP_PNAME addr_in[0]

#define AP_TBUFFER addr_in[0]
#define AP_TLENGTH int_in[0]
#define AP_TSCALE int_in[1]
```

```
#define SCR_MGR 0x0001                    /* pid of the screen manager*/

#define AP_MSG 0
#define MN_SELECTED 10

#define WM_REDRAW 20
#define WM_TOPPED 21
#define WM_CLOSED 22
#define WM_FULLED 23
#define WM_ARROWED 24
#define WM_HSLID 25
#define WM_VSLID 26
#define WM_SIZED 27
#define WM_MOVED 28
#define WM_NEWTOP 29

#define AC_OPEN 40
#define AC_CLOSE 41

#define CT_UPDATE 50
#define CT_MOVE 51
#define CT_NEWTOP 52
                                          /* event lib parameters */
#define IN_FLAGS int_in[0]

#define B_CLICKS int_in[0]
#define B_MASK int_in[1]
#define B_STATE int_in[2]

#define MO_FLAGS int_in[0]
#define MO_X int_in[1]
#define MO_Y int_in[2]
#define MO_WIDTH int_in[3]
#define MO_HEIGHT int_in[4]

#define ME_PBUFF addr_in[0]

#define T_LOCOUNT int_in[0]
#define T_HICOUNT int_in[1]

#define MU_FLAGS int_in[0]
#define EV_MX int_out[1]
#define EV_MY int_out[2]
#define EV_MB int_out[3]
#define EV_KS int_out[4]
#define EV_KRET int_out[5]
#define EV_BRET int_out[6]

#define MB_CLICKS int_in[1]
#define MB_MASK int_in[2]
#define MB_STATE int_in[3]
```

```
#define MM01_FLAGS int_in[4]
#define MM01_X int_in[5]
#define MM01_Y int_in[6]
#define MM01_WIDTH int_in[7]
#define MM01_HEIGHT int_in[8]

#define MM02_FLAGS int_in[9]
#define MM02_X int_in[10]
#define MM02_Y int_in[11]
#define MM02_WIDTH int_in[12]
#define MM02_HEIGHT int_in[13]

#define MME_PBUFF addr_in[0]

#define MT_LOCOUNT int_in[14]
#define MT_HICOUNT int_in[15]
                                        /* mu_flags                    */
#define MU_KEYBD 0x0001
#define MU_BUTTON 0x0002
#define MU_M1 0x0004
#define MU_M2 0x0008
#define MU_MESAG 0x0010
#define MU_TIMER 0x0020

#define EV_DCRATE int_in[0]
#define EV_DCSETIT int_in[1]
                                         /* menu library parameters */

#define MM_ITREE        addr_in[0]      /* ienable,icheck,tnorm */

#define MM_PSTR         addr_in[0]

#define MM_PTEXT        addr_in[1]

#define SHOW_IT         int_in[0]       /* bar                         */

#define ITEM_NUM        int_in[0]       /* icheck, ienable             */
#define MM_PID          int_in[0]       /* register                    */
#define CHECK_IT        int_in[1]       /* icheck                      */
#define ENABLE_IT       int_in[1]       /* ienable                     */

#define TITLE_NUM       int_in[0]       /* tnorm                       */
#define NORMRL_IT       int_in[1]       /* tnormal                     */

                                     /* form library parameters        */
#define FM_FORM addr_in[0]
#define FM_START int_in[0]

#define FM_TYPE int_in[0]

#define FM_ERRNUM int_in[0]
```

```
#define FM_DEFBUT int_in[0]
#define FM_ASTRING addr_in[0]

#define FM_IX int_in[1]
#define FM_IY int_in[2]
#define FM_IW int_in[3]
#define FM_IH int_in[4]
#define FM_X int_in[5]
#define FM_Y int_in[6]
#define FM_W int_in[7]
#define FM_H int_in[8]

#define FM_XC int_out[1]
#define FM_YC int_out[2]
#define FM_WC int_out[3]
#define FM_HC int_out[4]

#define FMD_START 0
#define FMD_GROW 1
#define FMD_SHRINK 2
#define FMD_FINISH 3
                                        /* object library parameters    */

#define OB_TREE addr_in[0]              /* all ob procedures            */

#define OB_DELOB int_in[0]              /* ob_delete                    */

#define OB_DRAWOB int_in[0]             /* ob_draw, ob_change            /
#define OB_DEPTH int_in[1]
#define OB_XCLIP int_in[2]
#define OB_YCLIP int_in[3]
#define OB_WCLIP int_in[4]
#define OB_HCLIP int_in[5]

#define OB_STARTOB int_in[0]            /* ob_find                      */
#define OB_DEPTH int_in[1]
#define OB_MX int_in[2]
#define OB_MY int_in[3]

#define OB_PARENT int_in[0]             /* ob_add                       */
#define OB_CHILD int_in[1]
#define OB_OBJ int_in[0]                /* ob_offset, ob_order          */
#define OB_XOFF int_out[1]
#define OB_YOFF int_out[2]
#define OB_NEWPOS int_in[1]             /* ob_order                     */

                                        /* ob_edit                      */
#define OB_CHAR int_in[1]
#define OB_IDX int_in[2]
#define OB_KIND int_in[3]
#define OB_ODX int_out[1]
```

```
#define OB_NEWSTATE int_in[6]          /* ob_change                    */
#define OB_REDRAW int_in[7]
                                       /* graphics library parameters  */
#define GR_I1 int_in[0]
#define GR_I2 int_in[1]
#define GR_I3 int_in[2]
#define GR_I4 int_in[3]
#define GR_I5 int_in[4]
#define GR_I6 int_in[5]
#define GR_I7 int_in[6]
#define GR_I8 int_in[7]

#define GR_O1 int_out[1]
#define GR_O2 int_out[2]

#define GR_TREE addr_in[0]
#define GR_PARENT int_in[0]
#define GR_OBJ int_in[1]
#define GR_INSTATE int_in[2]
#define GR_OUTSTATE int_in[3]

#define GR_ISVERT int_in[2]

#define M_OFF 256
#define M_ON 257

#define GR_MNUMBER int_in[0]
#define GR_MADDR addr_in[0]

#define GR_WCHAR int_out[1]
#define GR_HCHAR int_out[2]
#define GR_WBOX int_out[3]
#define GR_HBOX int_out[4]

#define GR_MX int_out[1]
#define GR_MY int_out[2]
#define GR_MSTATE int_out[3]
#define GR_KSTATE int_out[4]
                                       /* scrap library parameters     */
#define SC_PATH addr_in[0]
                                       /* file selector library parms  */

#define FS_IPATH addr_in[0]
#define FS_ISEL addr_in[1]

#define FS_BUTTON int_out[1]
                                       /* window library parameters    */
#define XFULL 0
#define YFULL gl_hbox
#define WFULL gl_width
#define HFULL (gl_height - gl_hbox)
```

```
#define NAME 0x0001
#define CLOSER 0x0002
#define FULLER 0x0004
#define MOVER 0x0008
#define INFO 0x0010
#define SIZER 0x0020
#define UPARROW 0x0040
#define DNARROW 0x0080
#define VSLIDE 0x0100
#define LFARROW 0x0200
#define RTARROW 0x0400
#define HSLIDE 0x0800

#define WF_KIND 1
#define WF_NAME 2
#define WF_INFO 3
#define WF_WXYWH 4
#define WF_CXYWH 5
#define WF_PXYWH 6
#define WF_FXYWH 7
#define WF_HSLIDE 8
#define WF_VSLIDE 9
#define WF_TOP 10
#define WF_FIRSTXYWH 11
#define WF_NEXTXYWH 12
#define WF_IGNORE 13
#define WF_NEWDESK 14
#define WF_HSLSIZ 15
#define WF_VSLSIZ 16
                                        /* arrow message                */
#define WA_UPPAGE 0
#define WA_DNPAGE 1
#define WA_UPLINE 2
#define WA_DNLINE 3
#define WA_LFPAGE 4
#define WA_RTPAGE 5
#define WA_LFLINE 6
#define WA_RTLINE 7
                                        /* wm_create                    */
#define WM_KIND int_in[0]
                                        /* wm_open, close, del          */
#define WM_HANDLE int_in[0]
                                        /* wm_open, wm_create           */
#define WM_WX int_in[1]
#define WM_WY int_in[2]
#define WM_WW int_in[3]
#define WM_WH int_in[4]
                                        /* wm_find                      */
#define WM_MX int_in[0]
#define WM_MY int_in[1]
                                        /* wm_calc                      */
#define WC_BORDER 0
#define WC_WORK 1
#define WM_WCTYPE int_in[0]
```

```
#define WM_WCKIND int_in[1]
#define WM_WCIX int_in[2]
#define WM_WCIY int_in[3]
#define WM_WCIW int_in[4]
#define WM_WCIH int_in[5]
#define WM_WCOX int_out[1]
#define WM_WCOY int_out[2]
#define WM_WCOW int_out[3]
#define WM_WCOH int_out[4]
                                        /* wm_update                    */
#define WM_BEGUP int_in[0]

#define WM_WFIELD int_in[1]

#define WM_IPRIVATE int_in[2]

#define WM_IKIND int_in[2]
                                        /* for name and info            */
#define WM_IOTITLE addr_in[0]

#define WM_IX int_in[2]
#define WM_IY int_in[3]
#define WM_IW int_in[4]
#define WM_IH int_in[5]

#define WM_OX int_out[1]
#define WM_OY int_out[2]
#define WM_OW int_out[3]
#define WM_OH int_out[4]

#define WM_ISLIDE int_in[2]

#define WM_IRECTNUM int_in[6]
                                        /* resource library parameters  */

#define RS_PFNAME addr_in[0]            /* rs_init,                     */
#define RS_TYPE int_in[0]
#define RS_INDEX int_in[1]
#define RS_INADDR addr_in[0]
#define RS_OUTADDR addr_out[0]

#define RS_TREE addr_in[0]
#define RS_OBJ int_in[0]

#define R_TREE 0
#define R_OBJECT 1
#define R_TEDINFO 2
#define R_ICONBLK 3
#define R_BITBLK 4
#define R_STRING 5
#define R_IMAGEDATA 6
#define R_OBSPEC 7
```

```
#define R_TEPTEXT 8             /* sub ptrs in TEDINFO  */
#define R_TEPTMPLT 9
#define R_TEPVALID 10
#define R_IBPMASK 11            /* sub ptrs in ICONBLK  */
#define R_IBPDATA 12
#define R_IBPTEXT 13
#define R_BIPDATA 14            /* sub ptrs in BITBLK   */
#define R_FRSTR 15              /* gets addr of ptr to free strings     */
#define R_FRIMG 16              /* gets addr of ptr to free images      */

                                /* shell library parametsrs     */
#define SH_DOEX int_in[0]
#define SH_ISGR int_in[1]
#define SH_ISCR int_in[2]
#define SH_PCMD addr_in[0]
#define SH_PTAIL addr_in[1]

#define SH_PDATA addr_in[0]
#define SH_PBUFFER addr_in[0]

#define SH_LEN int_in[0]

#define SH_PATH addr_in[0]
#define SH_SRCH addr_in[1]
```

GEMDEFS.H

```
/*********************************************************************/
/*      GEMDEFS.H Common GEM definitions and miscellaneous structures.  */
/*              Copyright 1985 Atari Corp.                              */
/*********************************************************************/
/*      EVENT Manager Definitions                                       */
                                                /* multiflags           */
#define MU_KEYBD  0x0001
#define MU_BUTTON 0x0002
#define MU_M1     0x0004
#define MU_M2     0x0008
#define MU_MESAG  0x0010
#define MU_TIMER  0x0020
                                                /* keyboard states      */
#define K_RSHIFT 0x0001
#define K_LSHIFT 0x0002
#define K_CTRL   0x0004
#define K_ALT    0x0008
                                                /* message values       */
#define MN_SELECTED 10
#define WM_REDRAW   20
#define WM_TOPPED   21
#define WM_CLOSED   22
#define WM_FULLED   23
#define WM_ARROWED  24
#define WM_HSLID    25
#define WM_VSLID    26
#define WM_SIZED    27
#define WM_MOVED    28
```

```
#define WM_NEWTOP   29
#define AC_OPEN     40
#define AC_CLOSE    41

/*      FORM Manager Definitions                                        */
                                                /* Form flags           */
#define FMD_START  0
#define FMD_GROW   1
#define FMD_SHRINK 2
#define FMD_FINISH 3

/*      RESOURCE Manager Definitions                                    */
                                                /* data structure types */
#define R_TREE     0
#define R_OBJECT   1
#define R_TEDINFO  2
#define R_ICONBLK  3
#define R_BITBLK   4
#define R_STRING   5            /* gets pointer to free strings */
#define R_IMAGEDATA 6           /* gets pointer to free images  */
#define R_OBSPEC   7
#define R_TEPTEXT  8            /* sub ptrs in TEDINFO  */
#define R_TEPTMPLT 9
#define R_TEPVALID 10
#define R_IBPMASK  11           /* sub ptrs in ICONBLK  */
#define R_IBPDATA  12
#define R_IBPTEXT  13
#define R_BIPDATA  14           /* sub ptrs in BITBLK   */
#define R_FRSTR    15           /* gets addr of ptr to free strings     */
#define R_FRIMG    16           /* gets addr of ptr to free images      */

                                                /* used in RSCREATE.C   */
typedef struct rshdr
{
        int             rsh_vrsn;
        int             rsh_object;
        int             rsh_tedinfo;
        int             rsh_iconblk;     /* list of ICONBLKS            */
        int             rsh_bitblk;
        int             rsh_frstr;
        int             rsh_string;
        int             rsh_imdata;     /* image data                   */
        int             rsh_frimg;
        int             rsh_trindex;
        int             rsh_nobs;       /* counts of various structs    */
        int             rsh_ntree;
        int             rsh_nted;
        int             rsh_nib;
        int             rsh_nbb;
        int             rsh_nstring;
        int             rsh_nimages;
        int             rsh_rssize;     /* total bytes in resource      */
} RSHDR;
#define F_ATTR 0                        /* file attr for dos_create     */
```

```
/*      WINDOW Manager Definitions                                      */
                                                        /* Window Attributes    */
#define NAME    0x0001
#define CLOSER  0x0002
#define FULLER  0x0004
#define MOVER   0x0008
#define INFO    0x0010
#define SIZER   0x0020
#define UPARROW 0x0040
#define DNARROW 0x0080
#define VSLIDE  0x0100
#define LFARROW 0x0200
#define RTARROW 0x0400
#define HSLIDE  0x0800
                                                        /* wind_create flags    */
#define WC_BORDER 0
#define WC_WORK   1
                                                        /* wind_get flags       */
#define WF_KIND         1
#define WF_NAME         2
#define WF_INFO         3
#define WF_WORKXYWH     4
#define WF_CURRXYWH     5
#define WF_PREVXYWH     6
#define WF_FULLXYWH     7
#define WF_HSLIDE       8
#define WF_VSLIDE       9
#define WF_TOP          10
#define WF_FIRSTXYWH    11
#define WF_NEXTXYWH     12
#define WF_RESVD        13
#define WF_NEWDESK      14
#define WF_HSLSIZE      15
#define WF_VSLSIZE      16
#define WF_SCREEN       17
                                                        /* update flags         */
#define END_UPDATE 0
#define BEG_UPDATE 1
#define END_MCTRL  2
#define BEG_MCTRL  3

/*      GRAPHICS Manager Definitions                                    */
                                                        /* Mouse Forms          */
#define ARROW       0
#define TEXT_CRSR   1
#define HOURGLASS   2
#define POINT_HAND  3
#define FLAT_HAND   4
#define THIN_CROSS  5
#define THICK_CROSS 6
#define OUTLN_CROSS 7
#define USER_DEF  255
#define M_OFF     256
#define M_ON      257
```

```
/*      MISCELLANEOUS Structures                                        */
                                                /* Memory Form Definition Block */
typedef struct fdbstr
{
        long    fd_addr;
        int             fd_w;
        int             fd_h;
        int             fd_wdwidth;
        int             fd_stand;
        int             fd_nplanes;
        int             fd_r1;
        int             fd_r2;
        int             fd_r3;
} FDB;
                                                /* Mouse Form Definition Block  */
typedef struct mfstr
{
        int     mf_xhot;
        int     mf_yhot;
        int     mf_nplanes;
        int     mf_fg;
        int     mf_bg;
        int     mf_mask[16];
        int     mf_data[16];
} MFORM;
```

OBDEFS.H

```
#define ROOT 0

#define MAX_LEN 81              /* max string length */

#define MAX_DEPTH 8             /* max depth of search or draw */

#define IP_HOLLOW 0             /* inside patterns */
#define IP_1PATT 1
#define IP_2PATT 2
#define IP_3PATT 3
#define IP_4PATT 4
#define IP_5PATT 5
#define IP_6PATT 6
#define IP_SOLID 7

#define MD_REPLACE 1    /* gsx modes */
#define MD_TRANS   2
#define MD_XOR     3
#define MD_ERASE   4

#define ALL_WHITE  0    /* bit bit rules */
#define S_AND_D    1
#define S_AND_NOTD 2
#define S_ONLY     3
#define NOTS_AND_D 4
#define D_ONLY     5
#define S_XOR_D    6
```

```
#define S_OR_D      7
#define NOT_SORD    8
#define NOT_SXORD   9
#define D_INVERT   10
#define NOT_D      11
#define S_OR_NOTD  12
#define NOTS_OR_D  13
#define NOT_SANDD  14
#define ALL_BLACK  15

#define IBM 3                    /* font types */
#define SMALL 5

#define G_BOX      20    /* Graphic types of obs */
#define G_TEXT     21
#define G_BOXTEXT  22
#define G_IMAGE    23
#define G_PROGDEF  24
#define G_IBOX     25
#define G_BUTTON   26
#define G_BOXCHAR  27
#define G_STRING   28
#define G_FTEXT    29
#define G_FBOXTEXT 30
#define G_ICON     31
#define G_TITLE    32

#define NONE       0x0  /* Object flags */
#define SELECTABLE 0x1
#define DEFAULT    0x2
#define EXIT       0x4
#define EDITABLE   0x8
#define RBUTTON   0x10
#define LASTOB    0x20
#define TOUCHEXIT 0x40
#define HIDETREE  0x80
#define INDIRECT 0x100

#define NORMAL     0x0   /* Object states */
#define SELECTED   0x1
#define CROSSED    0x2
#define CHECKED    0x4
#define DISABLED   0x8
#define OUTLINED  0x10
#define SHADOWED  0x20

#define WHITE      0               /* Object colors */
#define BLACK      1
#define RED        2
#define GREEN      3
#define BLUE       4
#define CYAN       5
#define YELLOW     6
```

```
#define MAGENTA  7
#define LWHITE   8
#define LBLACK   9
#define LRED     10
#define LGREEN   11
#define LBLUE    12
#define LCYAN    13
#define LYELLOW  14
#define LMAGENTA 15

#define EDSTART 0        /* editable text field definitions */
#define EDINIT  1
#define EDCHAR  2
#define EDEND   3

#define TE_LEFT  0      /* editable text justification */
#define TE_RIGHT 1
#define TE_CNTR  2

/*      Structure Definitions */

typedef struct object
{
        int             ob_next;        /* -> object's next sibling     */
        int             ob_head;        /* -> head of object's children */
        int             ob_tail;        /* -> tail of object's children */
        unsigned int ob_type;   /* type of object- BOX, CHAR,... */
        unsigned int ob_flags;  /* flags */
        unsigned int cb_state;  /* state- SELECTED, OPEN, ... */
        char *ob_spec;          /* 'out'- -> anything else */
        int             ob_x;           /* upper left corner of object */
        int             ob_y;           /* upper left corner of object */
        int             ob_width;       /* width of obj */
        int             ob_height;      /* height of obj */
} OBJECT;

typedef struct orect
{
        struct orect *o_link;
        int     o_x;
        int     o_y;
        int     o_w;
        int     o_h;
} ORECT;

typedef struct grect
{
        int     g_x;
        int     g_y;
        int     g_w;
        int     g_h;
} GRECT;
```

```
typedef struct text_edinfo
{
        char *te_ptext;         /* ptr to text (must be 1st) */
        char *te_ptmplt;        /* ptr to template */
        char *te_pvalid;        /* ptr to validation chrs. */
        int te_font;            /* font */
        int te_junk1;           /* junk word */
        int te_just;            /* justification- left, right... */
        int te_color;           /* color information word */
        int te_junk2;           /* junk word */
        int te_thickness;       /* border thickness */
        int te_txtlen;          /* length of text string */
        int te_tmplen;          /* length of template string */
} TEDINFO;

typedef struct icon_block
{
        int *ib_pmask;
        int *ib_pdata;
        char *ib_ptext;
        int ib_char;
        int ib_xchar;
        int ib_ychar;
        int ib_xicon;
        int ib_yicon;
        int ib_wicon;
        int ib_hicon;
        int ib_xtext;
        int ib_ytext;
        int ib_wtext;
        int ib_htext;
} ICONBLK;

typedef struct bit_block
{
        int *bi_pdata;          /* ptr to bit forms data */
        int     bi_wb;                  /* width of form in bytes */
        int     bi_hl;                  /* height in lines */
        int     bi_x;                   /* source x in bit form */
        int     bi_y;                   /* source y in bit form */
        int     bi_color;               /* fg color of bit */
} BITBLK;

typedef struct appl_blk
{
        int (*ub_code)();
        long ub_parm;
} APPLBLK;

typedef struct parm_blk
{
        OBJECT *pb_tree;
        int     pb_obj;
```

```
        int     pb_prevstate;
        int     pb_currstate;
        int     pb_x, pb_y, pb_w, pb_h;
        int     pb_xc, pb_yc, pb_wc, pb_hc;
        long pb_parm;
} PARMBLK;
```

STDIO.H

```
#ifndef _BUFSIZE

#define _BUFSIZE 512
#define _NFILE 73

typedef struct _iobuf {
    char *_ptr;
    int _cnt;
    char *_base;
    int _flag;
    int _fd;
    long _mark;   /* position relative to start of file of _base */
    int _bufsize;       /* buffer size for this file */
} FILE;
extern FILE _iob[_NFILE];

#define stdin (&_iob[0])
#define stdout (&_iob[1])
#define stderr (&_iob[2])

#define _READ    01
#define _WRITE   02
#define _APPEND  04
#define _UNBUF   010
#define _BIGBUF  020
#define _LINBUF  0400
#define _EOF     040
#define _ERR     0100
#define _DIRTY   0200   /* buffer was changed */

#define NULL 0L    /* must be long since it can be passed as a parameter */
#define EOF (-1)

#define getc(p) (--(p)->_cnt >= 0 ? *(p)->_ptr++ & 0377 : _fillbuf(p))
#define getchar() getc(stdin)
#define putc(x,p) (--(p)->_cnt >= 0 ? (*(p)->_ptr++ = (x)) & 0377 : \
    _flushbuf((x),p))
#define putchar(x) putc(x,stdout)
#define feof(p) ((p)->_flag&_EOF)
#define ferror(p) ((p)->_flag&_ERR)
#define clearerr(p) ((p)->_flag &= ~(_ERR | _EOF))
#define fileno(p) ((p)->_fd)
#define abs(x) ((x)<0?-(x):(x))
#define rand() (int)(_seed = _seed * 6907 + 130253)
#define srand(x) _seed = x;
```

```
extern FILE *fopen();
extern long ftell();
extern char *gets();
extern char *fgets();
extern char *malloc(), *calloc();
extern long _seed;

typedef long jmp_buf[9];

#endif
```

CTYPE.H

```
#ifndef _tolower

#define _tolower(c) c+32
#define _toupper(c) c-32
#define isalpha(c) (c>='A' && c<='Z' || c>='a' && c<='z')
#define isupper(c) (c>='A' && c<='Z')
#define islower(c) (c>='a' && c<='z')
#define isdigit(c) (c>='0' && c<='9')
#define isalnum(c) (isalpha(c) || isdigit(c))
#define isspace(c) (c==' ' || c=='\t' || C==' ' || c=='\n' || c=='\f')
#define ispunct(c) (c>=' ' && !isalnum(c))
#define isprint(c) (c>=040 && c<=0176)
#define iscntrl(c) (c>=0 && (c==0177 || c<' '))
#define isascii(c) (c>=0 && c<0200)

#endif
```

ERRNO.H

```
extern int errno;   /* defined in exit.c */
```

APPENDIX C

Keycode Values

In text-based computer environments, the computer and operating system would encode each alphanumeric character with a particular value. Whether in a data file, on a communication line, or from the keyboard, a particular value could be interpreted as the appropriate character. The most popular mapping of numbers to characters for personal computers is called ASCII (American Standard Code for Information Interchange). ASCII uses 7 of the 8 bits in a byte. This defines 128 characters for use by the computer.

Recently, many personal computers (including the Atari) have defined graphic characters to the remaining 128 characters that can be represented by one byte (one byte can represent 256 different values). Unfortunately, these characters are generally a hodge-podge of pictures like smiling faces, Greek or Hebrew letters, and so on, and they are hardly standardized from one computer to the next. On top of this, there are now more than 256 possible keystrokes that can be pressed by the user, such as combinations with the Control, Shift, and Alternate keys with the rest of the keyboard.

To alleviate this problem of understanding the data from the user, the GEM VDI defines a set of 16-bit values to represent the keystrokes entered by the user. The 16 bits are divided into two bytes. If the character is part of the ASCII set, the low byte contains the ASCII value. Otherwise it has the value of 0. For example, the letter "A" has an ASCII value of 65, so its low byte has the value of 65. The high byte contains an arbitrary value assigned by the VDI to differentiate the various keys on the keyboard. Below is a table containing the high and low byte values and the keystrokes required to create this keycode. The values shown are in hexadecimal.

GEM VDI Standard Keycode Values

High Byte	*Low Byte*	*Keystroke*
03	00	Control 2
1E	01	Control A
30	02	Control B
2E	03	Control C
20	04	Control D
12	05	Control E
21	06	Control F
22	07	Control G
23	08	Control H
17	09	Control I
24	0A	Control J
25	0B	Control K
26	0C	Control L
32	0D	Control M
31	0E	Control N
18	0F	Control O
19	10	Control P
10	11	Control Q
13	12	Control R
1F	13	Control S
14	14	Control T
16	15	Control U
2F	16	Control V
11	17	Control W
2D	18	Control X
15	19	Control Y
2C	1A	Control Z
1A	1B	Control [
2B	1C	Control \
1B	1D	Control]
07	1E	Control 6
0C	1F	Control –
39	20	Space
02	21	!
28	22	"
04	23	#
05	24	$
06	25	%
08	26	&
28	27	'
0A	28	(
0B	29	)
09	2A	*
0D	2B	<
33	2C	,
0C	2D	>
34	2E	.
35	2F	/

(*continued*)

High Byte	*Low Byte*	*Keystroke*
0B	30	0
02	31	1
03	32	2
04	33	3
05	34	4
06	35	5
07	36	6
08	37	7
09	38	8
0A	39	9
27	3A	:
27	3B	;
33	3C	{
0D	3D	=
34	3E	}
35	3F	?
03	40	@
1E	41	A
30	42	B
2E	43	C
20	44	D
12	45	E
21	46	F
22	47	G
23	48	H
17	49	I
24	4A	J
25	4B	K
26	4C	L
32	4D	M
31	4E	N
18	4F	O
19	50	P
10	51	Q
13	52	R
1F	53	S
14	54	T
16	55	U
2F	56	V
11	57	W
2D	58	X
15	59	Y
2C	5A	Z
1A	5B	[
2B	5C	\
1B	5D	]
07	5E	^
0C	5F	_ (Underscore)
29	60	`

(*continued*)

High Byte	*Low Byte*	*Keystroke*
1E	61	a
30	62	b
2E	63	c
20	64	d
12	65	e
21	66	f
22	67	g
23	68	h
17	69	i
24	6A	j
25	6B	k
26	6C	l
32	6D	m
31	6E	n
18	6F	o
19	70	p
10	71	q
13	72	r
1F	73	s
14	74	t
16	75	u
2F	76	v
11	77	w
2D	78	x
15	79	y
2C	7A	z
1A	7B	{
2B	7C	|
1B	7D	}
29	7E	~
0E	7F	Delete
81	00	Alternate 0
78	00	Alternate 1
79	00	Alternate 2
7A	00	Alternate 3
7B	00	Alternate 4
7B	00	Alternate 5
7D	00	Alternate 6
7E	00	Alternate 7
7F	00	Alternate 8
80	00	Alternate 9
1E	00	Alternate A
30	00	Alternate B
2E	00	Alternate C
20	00	Alternate D
12	00	Alternate E
21	00	Alternate F
22	00	Alternate G
23	00	Alternate H

(*continued*)

High Byte	*Low Byte*	*Keystroke*
17	00	Alternate I
24	00	Alternate J
25	00	Alternate K
26	00	Alternate L
32	00	Alternate M
31	00	Alternate N
18	00	Alternate O
19	00	Alternate P
10	00	Alternate Q
13	00	Alternate R
1F	00	Alternate S
14	00	Alternate T
16	00	Alternate U
2F	00	Alternate V
11	00	Alternate W
2D	00	Alternate X
15	00	Alternate Y
2C	00	Alternate Z
3B	00	F1
3C	00	F2
3D	00	F3
3E	00	F4
3F	00	F5
40	00	F6
41	00	F7
42	00	F8
43	00	F9
44	00	F10
54	00	F11
55	00	F12
56	00	F13
57	00	F14
58	00	F15
59	00	F16
5A	00	F17
5B	00	F18
5C	00	F19
5D	00	F20
5E	00	F21
5F	00	F22
60	00	F23
61	00	F24
62	00	F25
63	00	F26
64	00	F27
65	00	F28
66	00	F29
67	00	F30
68	00	F31

(*continued*)

High Byte	*Low Byte*	*Keystroke*
69	00	F32
6A	00	F33
6B	00	F34
6C	00	F35
6D	00	F36
6E	00	F37
6F	00	F38
70	00	F39
71	00	F40
73	00	Control left arrow
4D	00	Right arrow
4D	36	Shift right arrow
74	00	Control right arrow
50	00	Down arrow
50	32	Shift down arrow
48	00	Up arrow
48	38	Shift up arrow
51	00	Page down
51	33	Shfit page down
76	00	Control page down
49	00	Page up
49	39	Shift page up
84	00	Control page up
77	00	Control home
47	00	Home
47	37	Shift home
52	00	Insert
52	30	Shift insert
53	00	Delete
53	2E	Shift delete
72	00	Control print screen
37	2A	Print screen
01	1B	Escape
0E	08	Backspace
82	00	Alternate −
83	00	Alternate +
1C	0D	Carriage return
1C	0A	Control carriage return
4C	35	Shift numeric pad 5
4A	2B	Numeric pad −
4E	2B	Numeric pad +
0F	09	Tab
0F	00	Backtab
4B	00	Left arrow
4B	34	Shift left arrow
4F	00	End
4F	31	Shift end
75	00	Control end

APPENDIX D

System Variables

The Atari ST Basic Input/Output System (BIOS) uses a set of variables that define various characteristics of the computer. These variables reside at specific locations in memory. Atari has guaranteed that these memory locations will not change with future revisions to the ST BIOS. Therefore, if a program needs to access this information, it will always be able to find it at the same location on any ST machine. In order to read to or write from these addresses, the program must be in Supervisor mode.

The list below gives the address, length, name, and description of each location. The name is the common variable name used to reference this location. For convenience, most programmers use the variable names given here when working with the particular location.

NOTE: The information presented here is for completeness and general reference. Those readers who do not have experience in working with operating systems should avoid changing any of these values. If you want to use any of these variables, you should obtain the complete Atari ST documentation before continuing.

System Variables

etv_timer long 0x400

System Timer interrupt vector. Every 50 Hz, the routine pointed to by this value is called to maintain the system's date and time of day. This is the same vector as the GEMDOS logical vector 0x100, Timer Tick.

etv_critic long 0x404

Critical Error Handler vector. This location points to an error handling routine for certain error (e.g., disk errors and media changes). This is the same vector as the GEMDOS logical vector 0x101, Critical Error Handler.

etv_term long 0x408

Process Terminate vector. The routine pointed to by this value is called when a process terminates. This is the same vector as the GEMDOS logical vector 0x102, Terminate Handler.

etv_xtra 5 long 0x40C

Each long value is a pointer to a routine corresponding to the GEMDOS logical vectors 0x103 through 0x107. These vectors are reserved for use by later versions of GEMDOS.

memvalid long 0x420

This location holds the value 0x752019F3. This value is used in conjunction with location **memval2** to verify a successful cold start (that is, power on or pushing the reset button).

memcntlr byte 0x424

The value at this address contains the value used to configure the memory controller. The low four bits at this location specify the memory layout. Some common values are:

Memory size	*Value*
128K	0
512K	4
256K (2 banks)	0
1MB (2 banks)	5

resvalid long 0x426

If this location contains the number 0x31415926 on a system RESET, system execution jumps to the location in **resvector.**

resvector long 0x42A

System RESET trap vector. See description for **resvalid.**

phystop long 0x42E

The value here is the address of the physical end of RAM. This is the address of the first unusable byte (e.g., 0x80000 on a 512K machine).

_membot long 0x432

This location contains the address of the bottom of available memory. The GEM BIOS **Getmpb( )** function uses this value as the start of the transient program areas (TPA).

_memtop long 0x436

Top of available memory. The GEM BIOS **Getmpb( )** function uses this value as the end of the TPA.

memval2 long 0x43A

This location contains the number 0x237698AA. See **memvalid.**

flock WORD 0x43E

Locks usage of the direct memory access (DMA) chip. A nonzero value indicates that the DMA is in use.

seekrate WORD 0x440

Default floppy disk seek rate. Only bits 0 and 1 are used. They have the following meanings:

Bits	*Time*
00	6 ms
01	12 ms
10	2 ms
11	3 ms (default)

_timer_ms WORD 0x442

System timer calibration in milliseconds (ms). This value is usually set a 0x14 (20 decimal). The value is returned by the GEM BIOS function **tickcal( ).**

_fverify WORD 0x444

Floppy disk verify flag. A zero value here means that no verification is done for writing operations. A nonzero value means that all write operations to floppy disks are verified through a read. Verification on is the default.

_bootdev WORD 0x446

The number of the device used to boot the system.

palmode WORD 0x448

Video mode flag. A nonzero value indicates that the PAL mode (50 Hz video) is in use. A zero value means that the NTSC mode (60 Hz video) is in use.

defshiftmd byte 0x44A

This value indicates the color resolution to use if the system is forced to switch from a monochrome monitor to a color monitor.

sshiftmd WORD 0x44C

This value is equal to the current setting for the screen resolution hardware register. The values have the following interpretations:

0	320 x 200 x 4 (low resolution color)
1	640 x 200 x 2 (medium resolution color)
2	640 x 400 x 1 (high resolution monochrome)

_v_base_ad long 0x44E

Address of the physical screen memory. This value points to a continuous block of 32,000 bytes that starts on a half-page (256 byte) boundary.

vblsem WORD 0x452

A semaphore (mutual exclusion flag) to ensure that only one process at a time uses the vertical-blank interrupt handler. A value of one allows vertical blank processing.

nvbls WORD 0x454

Number of pointers in the vertical-blank queue. This value is set to 8 on system RESET.

_vblqueue long 0x456

This variable points to a list of pointers to routines used as vertical-retrace handlers. All of the handlers are executed at each vertical retract interrupt.

colorptr long 0x45A

A pointer to a color palette. This variable points to an array of 16 WORDs that define the color palette. The array is loaded during the next vertical retrace. If **colorptr** contains the NULL value, no palette is loaded. A NULL is placed into colorptr after the palette is loaded.

screenptr long 0x45E

The address of the new physical screen memory. During the next vertical retrace, the address in screenpt is loaded into **_v_bas_ad**, then a NULL is placed into **screenpt**. A NULL value indicates no change to **_v_bas_ad**.

_vbclock long 0x462

A count of the number of vertical-blank interrupts that have occurred since the last system RESET.

_frclock long 0x466

The number of vertical retrace interrupts that were processed (i.e., not blocked by **vblsem**).

hdv_init long 0x46A

A pointer to the hard disk initialization routine. This will be NULL if it is not used.

swv_vec long 0x46E

The address of a routine that is executed when the monitor is physically changed (i.e., when the monochrome monitor is plugged or unplugged). Default address is the system RESET routine.

hdv_bpb long 0x472

The address of a routine that returns the BIOS parameter block (BPB) for a hard disk. A NULL value indicates that this location is unused. This routine uses the same parameters and return values as the GEM BIOS function **Getbpb( )**.

hdv_rw long 0x476

The address to the routine that reads from and writes to the hard disk. A NULL indicates unused. The parameters and return values are the same as the **Rwabs( )** GEM BIOS function.

hdv_boot long 0x47A

The address of the routine to boot from the hard disk (NULL if unused).

hdv_mediach long 0x47E

The address of the routine to detect if a change in the hard disk's media. See GEM BIOS function **Mediach( )**.

_cmdload WORD 0x482

A nonzero value tells the system to try to load and execute the program COMMAND.PRG from the boot device. The boot sector can set this location to nonzero so that an application may be loaded instead of the GEM Desktop.

conterm byte 0x484

The bits in this location determine the console attributes as follows:

Bit	*Function*
0	1 = enable audible key click
1	1 = enable auto key repeat
2	1 = ring bell when ASCII value 7 is sent to CON:
3	1 = return the current value of kbshift in bits

24–31 when **Bconin( )** is called. See GEM BIOS function **Kbshift( )**.

themd long 0x48E

Address of a memory descriptor structure (struct MD). The memory descriptor defines the GEMDOS TPA limits which are set by a call to the GEM BIOS function **Getmpb( )**. Once GEMDOS has been initiated, the structure values may not be changed. See **Getmpb( )** for details on the memory descriptor structure.

savptr long 0x4A2

A pointer to an area in memory used to save the CPU registers during BIOS function calls.

_nflops WORD 0x4A6

The number of floppy disks physically attached to the system (0, 1, or 2).

sav_context long 0x4AE

Address of a memory location used to save the processor context state when a catastrophic error occurs (e.g., an odd address trap or divide by zero).

_bufl 2 longs 0x4B4

Two buffer control block (BCB) pointers. The first BCB contains information about the data sectors on the disk. The second BCB defines the sectors for the file allocation table (FAT) and the directory. A BCB has the following format:

```
struct BCB {
  struct BCB    *b_link;     /* next BCB */
  WORD          b_bufdrv;    /* drive # or -1 */
  WORD          b_buftyp;    /* buffer type */
  WORD          b_bufrec;    /* record # cached here */
  WORD          b_dirty;     /* dirty flag */
  DMD           *b_dm;       /* Drive Media Descriptor */
  char          *b_bufr;     /* the buffer itself */
};
```

The DMD structure is not defined in the GEM documentation.

_hz_200 long 0x4BC

This is the number of ticks from the 200 Hz timer. This value is divided by four to obtain the 50 Hz system timer.

the_env byte[4] 0x4BE

The default environment string initially set to four NULL characters.

_drvbits long 0x4C2

This is the value returned by the **Drvmap( )** GEM BIOS function.

_dskbufp long 0x4C6

The address of a 1,024-byte disk buffer.

_prt_cnt WORD 0x4EE

The number of times the Alternate-Help key combination has been pressed. This value is initially set at −1. A value of 0 initiates the screen dump routine to output the screen display to the printer. A value greater than 0 causes the screen dump routine to abort and resets **_prt_cnt** to −1.

_sysbase long 0x4F2

The address of the base of TOS. TOS may reside in ROM or RAM. If **_sysbase** is greater than **phystop**, TOS is in ROM.

_shell_p long 0x4F6

The address of data used by a shell.

end_os long 0x4FA

The address of the byte immediately following the last by used by TOS. This is the start of the TPA.

exec_os long 0x4FE

The address of a shell program. The shell program is executed by the BIOS once system initialization has completed. This address normally points to the AES.

Alphabetical Listing of Atari ST System Variables

_bootdev	WORD	0x446
_bufl	2 longs	0x4B4
_cmdload	WORD	0x482
colorptr	long	0x45A
conterm	byte	0x484
defshiftmd	byte	0x44A
_drvbits	long	0x4C2
_dskbufp	long	0x4C6
end_os	long	0x4FA
etv_critic	long	0x404
etv_term	long	0x408
etv_timer	long	0x400
etv_xtra	longs	0x40C
exec_os	long	0x4FE
flock	WORD	0x43E
_frclock	long	0x466

_fverify	WORD	0x444
hdv_boot	long	0x47A
hdv_bpb	long	0x472
hdv_init	long	0x46A
hdv_mediach	long	0x47E
hdv_rw	long	0x476
_hz_200	long	0x4BC
_membot	long	0x432
memcntlr	byte	0x424
_memtop	long	0x436
memval2	long	0x43A
memvalid	WORD	0x420
_nflops	WORD	0x4A6
nvbls	WORD	0x454
palmode	long	0x448
phystop	long	0x42E
_prt_cnt	WORD	0x4EE
resvalid	long	0x426
resvector	long	0x42A
sav_context	long	0x4AE
savptr	long	0x4A2
screenpt	long	0x45E
seekrate	WORD	0x440
_shell_p	long	0x4F6
sshiftmd	WORD	0x44C
swv_vec	long	0x46E
_sysbase	long	0x4F2
the_env	byte[4]	0x4BE
themd	long	0x4BE
_timr_ms	WORD	0x442
_v_base_ad	long	0x44E
_vbclock	long	0x462
_vblqueue	long	0x456
vblsem	WORD	0x452

APPENDIX E

Predefined Message Events

Because GEM is a multitasking operating environment, several applications are executing at any given time. Among these applications are routines contained in the Application Environment Services (AES) such as the dispatcher and screen manager. In a multitasking situation, a mechanism must be provided to allow for interaction between applications and between the user and the system. GEM provides this mechanism through messages.

An application may send a message to any other application. The receiving application is issued a message event to indicate that a message has arrived. When the receiving application acknowledges the message event, a 16-byte message buffer is sent to the receiving program. The message buffer consists of an eight WORD array and has the following format:

Element	*Use*
0	Message type
1	ID number of sending application
2	Message length in excess of 16 bytes
3–7	Data dependent upon message type

If a message has more than 16 bytes, the number of bytes remaining to be read is the value in element 2 of the message buffer. The remaining bytes must be read with a call to the AES function **appl_read( )**.

The AES has several predefined message types. Below is a list of these message types along with its defined constant name, a description, and the usage of the remaining message buffer elements.

AES Predefined Messages

Type	Constant

10 MN_SELECTED

This message is sent when a user selects an active menu item.
Element 3—the object index of the menu title selected.
Element 4—the object index of the menu item selected.

20 WM_REDRAW

This message indicates that some portion of the window's work area needs to be redrawn
Element 3—handle of the window to be redrawn
Element 4—x coordinate of the redraw area
Element 5—y coordinate of the redraw area
Element 6—width of the redraw area
Element 7—height of the redraw area

21 WM_TOPPED

This message tells an application that a window is to be moved to the top and made active.
Element 3—the handle of the window

22 WM_CLOSED

This message is sent when the user has requested the window to be closed.
Element 3—the handle of the window to close

23 WM_FULLED

When the user clicks the mouse inside a window's full box, this message is sent to the application owning the window.
Element 3—the handle of the window

24 WM_ARROWED

This message informs the application when a user has clicked the mouse in the arrow or scroll bar areas of the window.

Element 3—the handle of the window
Element 4—the control area used, as follows:

0 = page up
1 = page down
2 = row up
3 = row down
4 = page left
5 = page right
6 = column left
7 = column right

Page actions are caused by interactions with the scroll bars. Row and column actions are caused by the arrows.

25 WM_HSLID

This message informs the application of a new position requested for the horizontal slider.

Element 3—the handle of the window

Element 4—requested slider position (0–1000)

26 WM_VSLID

This message informs the application of a new position requested for the vertical slider.

Element 3—the handle of the window

Element 4—requested slider position (0–1000)

27 WM_SIZED

This message indicates that the user has requested a new window size. The coordinates given by this message include the border control areas for the window.

Element 3—the handle of the window

Element 4—the x coordinate of the window (usually the current coordinate)

Element 5—the y coordinate of the window (usually the current coordinate)

Element 6—the requested width

Element 7—the requested height

28 WM_MOVED

This message tells the application that the user has requested a new location for the window. The coordinates given by this message include the border control areas for the window.

Element 3—the handle of the window

Element 4—the requested x coordinate

Element 5—the requested y coordinate

Element 6—the requested width (should remain the same)

Element 7—the requested height (should remain the same)

29 WM_NEWTOP

This message indicates that one of the application's windows has been placed on top and is the new active window.

Element 3—the handle of the window

30 AC_OPEN

This message is sent to a desk accessory when the user has selected it from the Desk menu.

Element 3—the ID as returned by the **menu_register( )** call

31 AC_CLOSE

This message is sent to a desk accessory in any one of the following conditions:

the current application has just terminated
the screen is about to be cleared
the window manager structure are about to be reinitialized

The desk accessory should delete any window it owns from the AES.
Element 3—the ID as returned by the **menu_register( )** call

APPENDIX F

GEM BIOS and DOS Error Codes

All error numbers are negative and fall into two categories. The numbers ranging from −1 to −31 are BIOS errors, and numbers ranging from −32 to −127 are DOS errors.

BIOS Error Codes

Number	*Description*
0	OK (no error)
−1	Error (general)
−2	Drive not ready
−3	Unknown command
−4	CRC error
−5	Bad request
−6	Seek error
−7	Unknown media
−8	Sector not found
−9	Out of paper
−10	Write fault
−11	Read fault
−13	Write on write-protected media
−14	Media change detected
−15	Unknown device
−16	Bad sectors on format
−17	Insert other disk (request)

Error −17 is actually a request from the BIOS for another disk to be inserted into drive A. This allows GEMDOS to think it has two drives on a single drive system.

GEMDOS Error Codes

Number	*Description*
−32	Invalid function number
−33	File not found
−34	Path not found
−35	Handle pool exhausted
−36	Access denied
−37	Invalid handle
−39	Insufficient memory
−40	Invalid memory block address
−46	Invalid drive specification
−47	No more files
−64	Range error
−65	GEMDOS internal error
−66	Invalid executable file format
−67	Memory block growth failure

APPENDIX G

Listing for File EXTRA.C

The function **rc_intersect( )** is not part of the GEM system. Therefore it is not documented as part of GEM. However, it is used in the example programs for both the Megamax and the Atari development system. In case your C programming system does not include this function in its libraries, the source code is provided below. The file shown is the file used by the Megamax system. For a description of the function's usage, see Appendix A or Chapter 12.

```
#include   <obdefs.h>
#include   <gemdefs.h>
#include   <osbind.h>

#define    WORD    int

WORD    min(a, b)
WORD    a, b;
{
    return ((a < b) ? a : b);
}

WORD    max(a, b)
WORD    a, b;
{
    return ((a > b) ? a : b);
}

WORD   rc_intersect(p1, p2)
GRECT  *p1, *p2;
{
WORD   tx, ty, tw, th;
```

```
    tw = min (p2->g_x + p2->g_w, p1->g_x + p1->g_w);
    th = min (p2->g_y + p2->g_h, p1->g_y + p1->g_h);
    tx = max (p2->g_x, p1->g_x);
    ty = max (p2->g_y, p1->g_y);
    p2->g_x = tx;
    p2->g_y = ty;
    p2->g_w = tw - tx;
    p2->g_h = th - ty;
    return ((tw > tx) && (th > ty));
}
```

Index

B

C

D

H

I

J

K

L

M

N

O

P

Q

R

S

Atari ST Application Programming is:

the combination of Atari ST and Digital Research's Graphics Environment Manager (GEM) which provides everyone with a personal computer that is easy and intuitive to use. Now you can learn to write complete software programs for this dynamic duo of the computer world. With **Atari ST Application Programming** you will learn and understand the varied concepts associated with a graphics based user interface.

Not just a reference guide, this book takes you deep into the minds of the software engineers who built this system. You will see the basic building blocks from the Virtual Device Interface to the Application Environment Services to the multi-voiced sound effects. The book presents detailed sample programs that you can experiment with and learn from.

With **Atari ST Application Programming**, you can:

- See how to create complex graphic images.
- Work with windows and menus to make your program easier to use.
- Find out how to create lavish sound effects.
- Use the mouse to select, move, and size objects.
- Create resource files for elegant screen designs.
- Write a program that shows you the secrets of animation.
- Have an instant library of over a dozen fun and useful programs.

In clear, concise, and understandable language, this book gets your imagination working in high gear. Once you have finished reading this book, you'll be ready to write sophisticated and robust applications that rival even the commercial packages.

> *"We started with the very basics that a programmer must know in order to program this computer. On top of this, we built a set of tools that can help programmers reach beyond the limit of their abilities."* — LAWRENCE J. POLLACK AND ERIC J.T. WEBER

Lawrence J. Pollack, author of two other books: *Programming in C on the IBM PC* and *Programming the Macintosh in C*; a computerized medical management package; and a database management system. Eric J.T. Weber is an independent computer consultant who has written numerous programs and users documentation.

52495
9 780553 343977

N 0-553-34397-1>>2495

34397-1 ■ IN U.S. $24.95 (IN CANADA $29.95) ■ BANTAM COMPUTER BOOKS