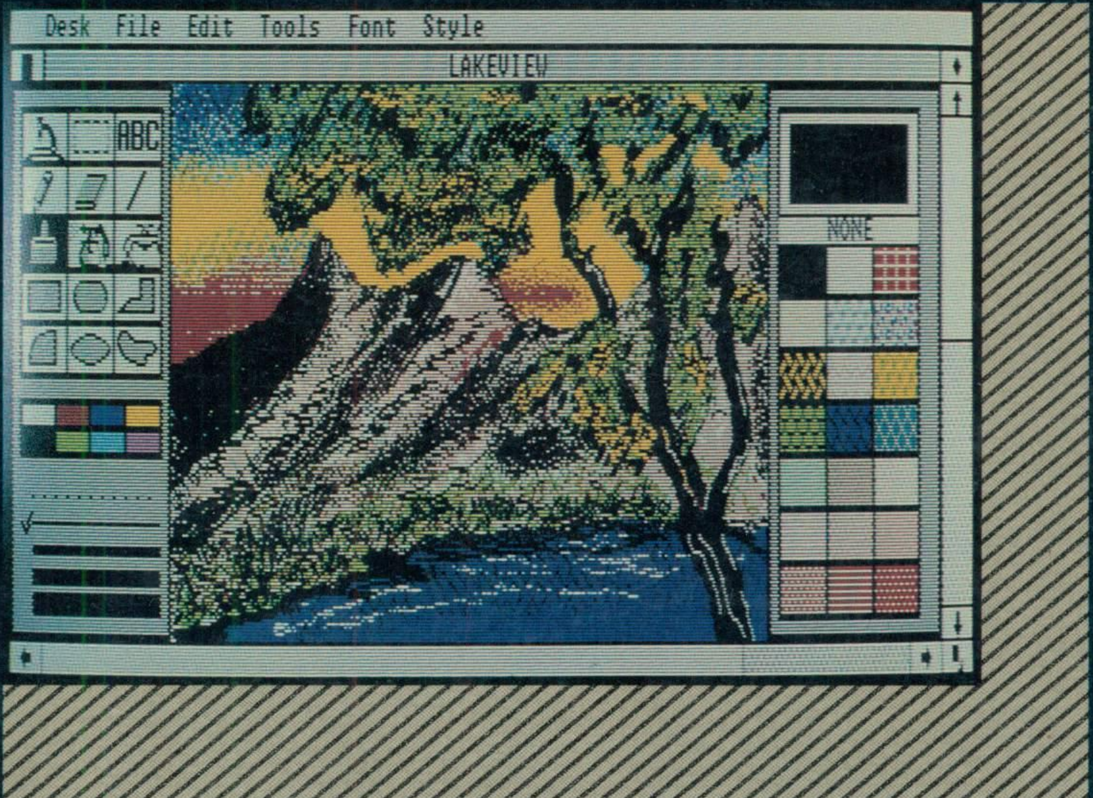
S
Y
B
E
X

PROGRAMMER'S GUIDE TO GEM™

Phillip Balma William Fitler



SYBEX

PROGRAMMER'S GUIDE TO GEM

PROGRAMMER'S GUIDE TO GEM™

PHILLIP BALMA

WILLIAM FITLER

SYBEX®

Berkeley • Paris • Düsseldorf • London

Cover art by Thomas Ingalls + Associates
Book design by Rick van Genderen

Library of Congress Card Number: 86-61242
ISBN 0-89588-297-3
Printed by Haddon Craftsmen
Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

ACKNOWLEDGMENTS

This book represents a collaboration of efforts. First and foremost, we would like to acknowledge our wives, Penny Fitler and Maxine Balma, for supporting us throughout this project. Next, we would like to thank our editor, Geta Carlson, whose patient and thorough efforts have greatly improved the quality of this book.

We have also appreciated the technical assistance of all of the members of the GEM team at Digital Research, including but not limited to Rich Greco, Lowell Webster, Rick Rosenbaum, Dan Brown, and Gregg Morris.

At SYBEX, we would like to thank Dan Tauber for technical edit, Olivia Shinomoto for word processing, Donna Scanlon for typesetting, and Aidan Wylde for proofreading.

CONTENTS

PREFACE

This book is an in-depth introduction to GEM for programmers. We assume that you know how to program and that you can at least read programs written in the C language. We have used a number of technical terms, along with a moderate amount of jargon peculiar to GEM. To help you with unfamiliar terminology, we've included a glossary of terms, as well as explanations in the text.

HOW THIS BOOK IS ORGANIZED

The first three chapters of this book will give you a thorough conceptual grounding in GEM and its two major components, the Application Environment Services and the Virtual Device Interface. The second three chapters put this information into practice by taking you step-by-step through the construction of two GEM applications and by discussing a variety of important design and coding issues. Six helpful appendixes supplement the information in the main body of the text. Here is a quick breakdown of what each chapter and appendix contains:

Chapter 1 is an overview of GEM. We discuss the various components of GEM and their interrelationships. We also talk about how GEM programs are built and organized.

Chapter 2 presents the Application Environment Services (AES). The chapter explains the important concepts used in the AES, including objects, events, and resources. It also discusses how these aspects of the AES are used to implement menus, forms, and alerts.

In Chapter 3 we cover functions provided by the Virtual Device Interface (VDI), which contains the functions that the AES and GEM applications use for input and output. We discuss the major strength of the VDI, which is to provide graphics display functions that are portable across a variety of different computers and hardware devices.

The last three chapters build upon this foundation. Chapter 4 presents the simplest GEM application that uses windows, called HELLO. Chapter 5 walks you through a much lengthier sample program called DEMO, which allows the user to create simple drawings on the screen. We rely on these programs quite a bit to demonstrate by example some of the more subtle points of developing GEM applications, as we have found that it is not enough simply to discuss the GEM functions and expect it to be obvious how to put them to use.

The purpose of Chapter 6 is to present a number of GEM program design considerations. Chapter 6 discusses several key design principles and explains how GEM functions can be used according to these principles. Chapter 6 also includes other important topics, such as how to use metafiles, which are relevant to GEM program design but did not really fit into the scheme of the previous chapters.

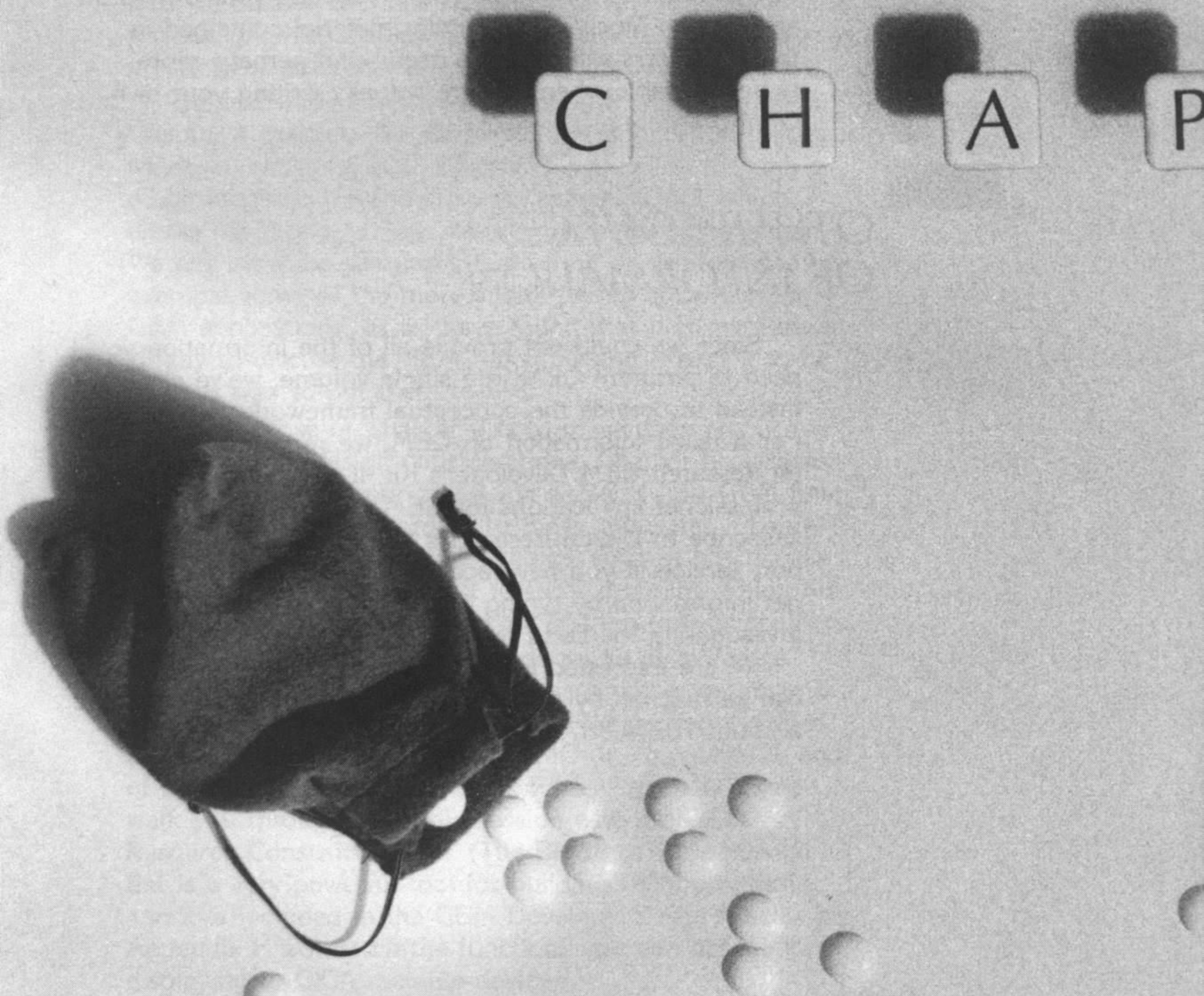
The appendixes contain a wide range of reference information. Appendix A is a glossary, while Appendix B provides a listing of GEM functions. We've also included the complete code listings to DEMO (in Appendix D) and of its resource file (in Appendix E). In Appendix C, we walk you through a sample session with the GEM Resource Construction Set. (The Resource Construction Set is a very powerful tool for building GEM programs, and it is included in the GEM Developer's Kit.) Finally, in Appendix F we list all the functions you can use when displaying to GEM metafile devices.

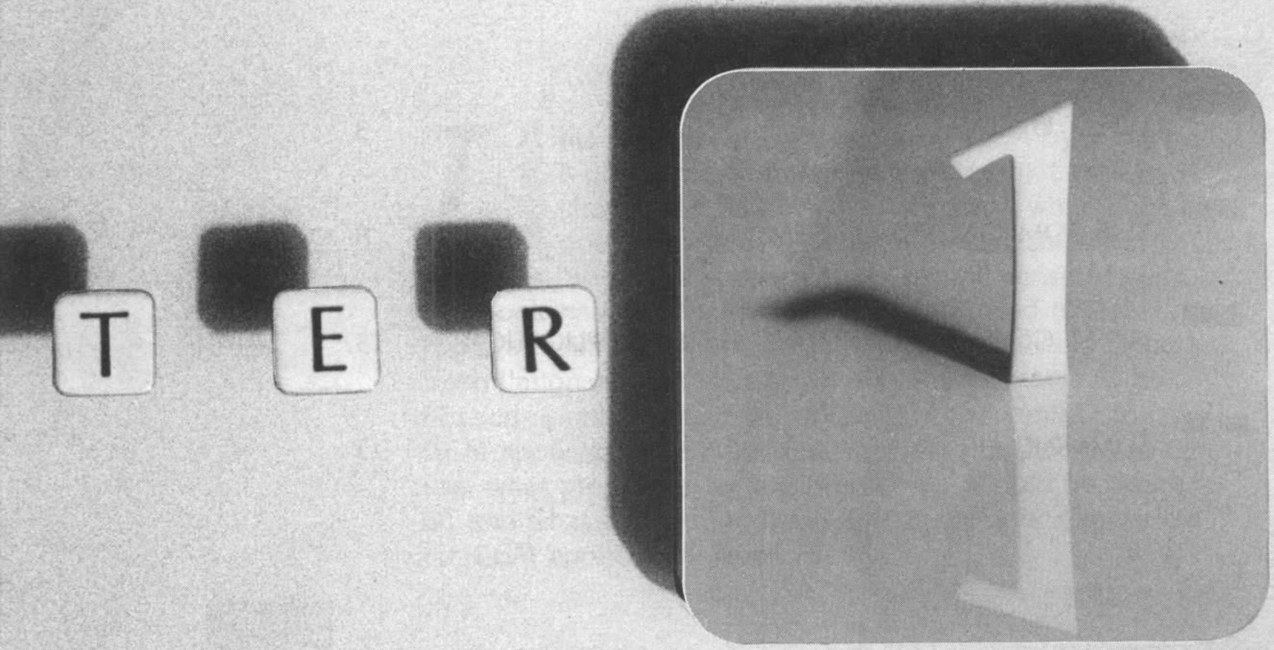
There are several ways to read this book, including the obvious method of reading cover to cover. You might want to read the reference chapters on the AES (Chapter 2) and the VDI (Chapter 3) somewhat quickly on the first reading, as well as the DEMO example (Chapter 5) and then refer to these chapters again as you are writing your own GEM application. Much of the subtler material contained in these chapters will be more useful and perhaps more understandable when you are actually writing your own GEM program.

OTHER SOURCES OF INFORMATION

Since we could not provide all of the information you'll need to program GEM in a single volume, we've aimed instead to provide the conceptual framework you'll need. For detailed information on GEM, we recommend the Digital Research GEM Developer's Kit. If you plan to develop professional applications for GEM, you'll probably want to subscribe to Digital Research's GEM Programmer's Support service. If you have access to CompuServe, you can get information by typing "GO DRI" at the prompt, and investigating the DRI Forum.

We are interested in your opinions about this book. We can be reached by sending electronic mail to CompuServe account 76244,367.


C H A P



INTRODUCTION TO GEM

This chapter introduces you to GEM by explaining its basic features. In the first part of this chapter, we will briefly summarize what GEM does and relate it to other similar software packages. We will also give you an overall sense of what this book has to offer and what you will need to use it. Next, we will focus on the components of GEM and their interrelationships as well as the way GEM relates to its host computer. Finally, we will present the context for GEM application development, including the tools you can use, what programming techniques you should be aware of, and what kind of hardware environment is appropriate for GEM application development.

WHAT IS GEM?

GEM is an operating environment. The word GEM is an acronym for *Graphics Environment Manager.* It provides a variety of high-level functions whose purpose is to make it easier for you, the applications programmer, to develop software that is both efficient and easy to use.

An *operating environment* is similar to an operating system. Whereas an operating system allows your program to utilize console and disk devices in a standard manner, the GEM operating environment allows your GEM program to control a number of graphics devices in a consistent and standard fashion. Thus, GEM promotes program portability across many different graphics devices.

Perhaps an even more important function of the GEM operating environment is its role in your program's interface to the user. If you think of the operating system as the way your program talks to the computer, you can think of the operating environment as the way your program talks to the user. Figure 1.1 shows this relationship between your program, the user, and the computer. With the GEM functions, your program can control the devices the user sees and manipulates, including the keyboard, the mouse and the screen and also the printer and the plotter, and even a film recorder capable of producing high-resolution slides. GEM is very similar to the operating system in that it allows you to write your program without having to worry about what kind of mouse is attached to the computer, what resolution the screen has, whether the computer's screen is color or monochrome, or most of the other myriad of differences between graphics devices.

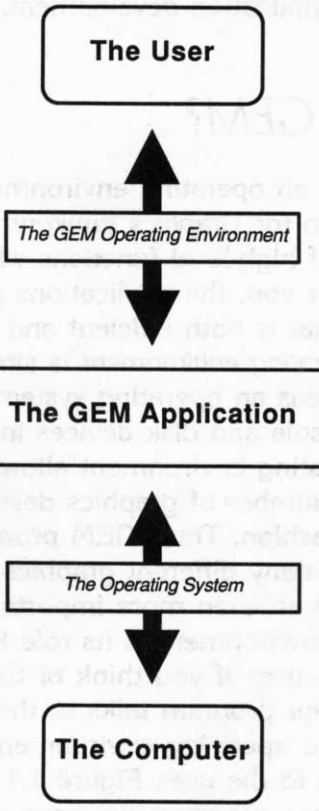


Figure 1.1: *The Role of the GEM Operating Environment*

Why Program with GEM?

One big reason for using GEM is that GEM is designed to help you write programs that work with many different graphics devices. It also facilitates making your program portable onto very different microcomputers. GEM is currently available for the IBM PC and compatible computers, running with the Intel 8086 microprocessor architecture. GEM is also available on the Atari ST, which uses the completely different Motorola 68000 microprocessor. We know of no other operating environment that is as portable as GEM. GEM's program portability helps to protect your software investment.

Another big advantage of GEM is the framework it provides to make GEM programs easy to use. In this way, it is similar to several other environments on the market today, including Apple's Macintosh, Microsoft's Windows, and IBM's TopView.

GEM programs are easy to use for several reasons. First, the visual effects of graphics can generally communicate information more effectively than text. The graphical images that your program uses can serve to suggest how your program works. For example, the graphical image of a folder suggests that it contains documents, drawings, and even other folders. Second, GEM provides standard controls that all GEM applications can use in a consistent fashion. For instance, one of the standard controls is the menu bar across the top of the screen. Once the user learns how to use the menus in one GEM program, she already knows how to operate the menu bar in your GEM program. The same holds true for manipulating GEM windows and for selecting files.

GEM provides some powerful functions that you can use to build very interactive applications. By "very interactive," we mean a type of user interface where a significant portion of the design and development effort goes into making the program easy to use.

Some of these functions deal with events. Events consists of anything from a key getting pressed on the keyboard, to a timer signalling that some amount of time has elapsed, to a message indicating that the user has selected a particular item from the menu. A particularly useful capability of GEM's is that it allows your program to wait for one or more of these events to occur. This means that your GEM program can make very efficient use of the computer by not running until one or more relevant events have occurred and by thus allowing other processes, such as desk accessories, to use the computer instead. Programs written for the Macintosh, for example, only look at single events. This limited multitasking capability of GEM's—that is, its ability to wait for multiple events—makes it easier to build very interactive programs.

It's important to note that although Digital Research may develop a true multi-application (as well as multitasking) version of GEM, the current version only supports a single application running at any given time along with a number of auxiliary programs (the desk accessories). Both Microsoft's Windows and IBM's TopView will support multiple concurrent applications, provided they are "well-behaved," which means that they don't directly manipulate the computer's hardware. Most popular applications, by the way, are not well-behaved.

GEM, on the other hand, was built for a single application, which has the very beneficial effect of allowing programs written for GEM to run, on the average, much faster than programs written for Windows or TopView. GEM also takes less RAM to run programs, which means that your GEM program will run on a greater range of IBM PCs and PC compatibles than a Windows or TopView program. Furthermore, GEM is available on other types of computers, such as the Atari ST. Currently, Windows and TopView are only available for the IBM PC and its clones.

If your program really needs multitasking, Digital Research provides other alternatives for applications requiring multitasking, most notably Concurrent PC-DOS (which, by the way, will run GEM programs). Contact Digital Research for more information about Concurrent PC-DOS.

What This Book Has To Offer

This book can be read by anyone curious about how the GEM operating environment is used to construct interactive graphical applications. Although GEM programs can be written in a number of different languages, including Pascal, Fortran, or assembler, in this book we have provided all of our examples in C, and we assume that you have a working knowledge of C. If you haven't programmed in C before, you will need a companion text on the C language. We also use certain conventions with our C programs to promote portability. We'll talk about these conventions in more detail later in this chapter.

You'll probably get the most out of this book if you've also acquired the GEM Developer's Kit, available from Digital Research or Atari. In fact, this book assumes you have access to a number of tools available in the kit if you plan to run and modify the examples.

This book is an ideal companion to the GEM Developer's Kit for several reasons. First, this book provides a more gentle and thorough introduction to the ins and outs of GEM than the Developer's Kit documentation provides. We've included a number of small sample programs that illustrate how to use GEM functions to create an application program. We've also taken two of the sample programs available in the Kit, HELLO and DEMO, and thoroughly revised and commented them. Finally, we've attempted to explain not only what the functions do, but also why they are built the way they are, along with their quirks, oddities, and some historical perspective. The principle that prompted and guided this book's creation is that it's easier to remember things when you understand them.

THE COMPONENTS OF GEM

Now that you have a rough idea of what GEM is and how it relates to other alternatives in the microcomputer software marketplace, let's explore the components of GEM in more detail. The purpose of this section is to give you an overall understanding of how GEM can be used by your program.

GEM consists of two major functional units: the *Application Environment Services (AES)* and the *Virtual Device Interface (VDI)*. The AES is further subdivided into a set of libraries, a limited multitasking kernel, and a Screen Manager. The VDI is composed of a *Graphics Device Operating System (GDOS)* that provides a complete set of basic graphics functions, and a set of drivers for graphics devices. While we will be discussing the details of each of these components in the next two chapters, we want to describe the basic roles of GEM's two subsystems here.

The Role of the VDI

The VDI provides one essential service for the programmer: a generalized logical interface between the programmer and any graphics device. A few years ago, each graphics device came with its own programming interface or set of libraries. When a programmer wanted to write a program that used Company X's graphics display, she would have to follow the conventions and syntax of the libraries of Company X. If the programmer then wanted to run this application on Company Y's machine, she would have to rewrite the application using Company Y's conventions.

Since no one who had to work in this area thought that this was a very a good system, a standard interface for all graphics devices was developed.

Unfortunately, however, there are several flavors of this standard. The real usefulness of a graphics standard is that a single program written to a standard set of graphics functions can be used on a variety of graphics devices without being rewritten, recompiled, or relinked. Such a one-to-many relationship between application and devices allows a programmer to write device-independent applications. The GEM VDI provides this support.

As part of the support for a true graphics standard, the GEM VDI provides a large set of drivers to make the final connection between the software and the hardware. Since each device has its own set of commands, the *driver* must translate the requests of the application into the hardware instructions that the device recognizes. Thus, for each different installed device on the system there must be a driver. The VDI's GDOS is responsible for making the connection between the application program and the specific device driver.

By the way, the term "virtual" means that now the programmer is writing her application to a *virtual graphics machine.* This virtual machine has all the characteristics of all the various graphics devices. It is a supermachine that is at once a plotter, a CRT device, a light pen, a mouse, and a graphics printer. For instance, the programmer can write an application that uses color, and it will still run on a monochrome display. The GEM VDI will interpret the application's request for color, recognize that the installed device does not support color, and make a suitable choice between black (for every color but white) or white.

The Role of the AES

One of the original goals of GEM's designers was to provide a graphical user interface to a computer. This interface is now known as the *GEM Desktop.* As Digital Research, Inc. (DRI) designed and built the Desktop, they designed and built a set of software routines that were needed by the Desktop. DRI gathered these routines into 11 software libraries, each categorized by their overall function. For instance, all the routines that manipulate windows were collected and form the Window Library of the AES. Similarly, all of the event routines form the Event Library, and so on. Thus, the AES represents a set of tools that was found useful in writing the first GEM application, the Desktop, and that now are useful in developing any GEM application.

The AES libraries are categorized by the kind of functions they deliver. There are libraries for window handling, event and resource management, and eight other categories of services. The AES Screen Manager helps to handle the mouse and the window mechanism, and the limited multitasking kernel coordinates the various independent tasks going on within GEM, including the Screen Manager itself, the desk accessories, and your program.

The kernel and Screen Manager are a part of the AES because they are essential for the event-handling system. They are not visible to the programmer, in that the service they provide is an action resulting from the use of the AES libraries and not something directly controlled by the programmer. In other words, your program does not communicate directly with either the kernel or the Screen Manager.

The library functions of the AES rely on basic graphics primitives in the VDI to construct and manipulate the features of applications, including windows, dialogs, and alerts. Thus, while the VDI provides routines to draw a line, display text, and accept input from the mouse and keyboard, for example, the AES uses those routines to provide functions that display a form on the screen device and allow the user to enter information into the fields of that form.

While the AES uses VDI functions, the VDI does not use any AES function. Thus, the AES can be viewed as sitting on top of the VDI, in a top-down hierarchy. A few GEM applications use the VDI exclusively and do not use the AES at all. We do not recommend this, mostly because it precludes the use of desk accessories, but also because we advocate using AES features like windows and menus to give a consistent user interface to GEM applications.

How GEM Works with the Operating System

The AES and VDI are two legs of the tripod on which GEM applications are built. The third leg is the operating system, as Figure 1.2 illustrates. Let's talk about the role of the operating system in GEM programs.

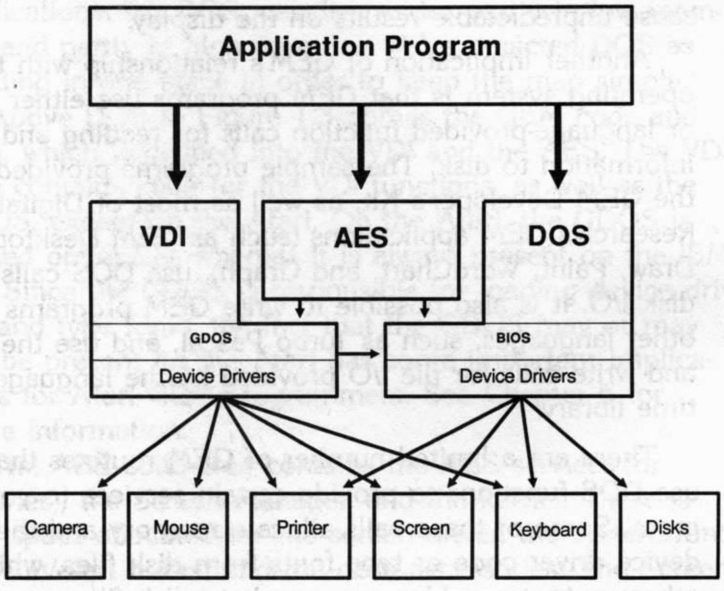


Figure 1.2: *The Relationship between the AES, VDI, and DOS*

Throughout this book, we use the generic term *DOS,* by which we mean IBM's *PC-DOS,* Microsoft's *MS-DOS,* Digital Research's *GEMDOS,* or Atari's *TOS.* Please note that GEM does not attempt to provide many of the services offered by DOS, such as file and memory management, but instead relies on the underlying operating system to provide these services.

This relationship has several implications for GEM programs that are important to understand. First and foremost is that your GEM program must never make any calls to the operating system's console input and output routines. Instead, all keyboard and mouse input and all screen output should be performed via GEM calls. The reason for this is that GEM needs complete control of these devices in order to consistently monitor and keep track of their internal state. If the screen is in graphics mode and your program performs character output, it can cause unpredictable results on the display.

Another implication of GEM's relationship with the operating system is that GEM programs use either DOS or language-provided function calls for reading and writing information to disk. The sample programs provided with the GEM Developer's Kit, as well as most of Digital Research's GEM applications (such as GEM Desktop, Draw, Paint, WordChart, and Graph), use DOS calls for disk I/O. It is also possible to write GEM programs in other languages, such as Turbo Pascal, and use the read and write calls for file I/O provided by the language runtime library.

There are a limited number of GEM routines that also use DOS functions to provide certain services to your program. Some of these calls allocate memory and then load device driver code or type fonts from disk files, while others output graphics commands to disk files, so that they can be saved and displayed at some later date.

Currently, GEM is available for PC-DOS and GEMDOS/TOS, and it will be available for other operating systems, such as Digital Research's Concurrent DOS 286. To maximize the portability of your program across different operating systems, keep all DOS calls localized to one or a few of your program's modules. This way, if the details of the operating system interface change, you will easily be able to locate the code that needs to be changed.

How GEM Is Laid Out in RAM

In order to illustrate how the components of GEM fit together, let's look at the way GEM uses memory. The information we present here is specific to GEM on the IBM PC, but we will also explain how GEM's use of memory differs on the Atari ST.

The functional layout of GEM and a GEM application on a PC is shown in Figure 1.3. The base for most PC applications, the DOS, usually resides partly in low memory and partly in high memory. We've depicted DOS as residing in one place in order to keep the map simple.

Above DOS in Figure 1.3 comes the GEM code and data, which is divided into the VDI and the AES. The VDI area contains code for the VDI functions, as well as the GDOS, except on the Atari. On the Atari, the GDOS is loaded optionally, whereas it is always present on the IBM PC. Since the GDOS is responsible for loading device drivers and type fonts, the fact that the GDOS may or may not be present on the Atari has some important implications for Atari GEM programmers. See Chapter 6 for more information.

The AES code area contains the AES services (or libraries), the Screen Manager, and the kernel. There is also space allocated for the screen driver, the system font, a menu/alert buffer, and the desk accessories. The Screen Manager requires the screen driver and system font, which means that these have already been loaded into the AES

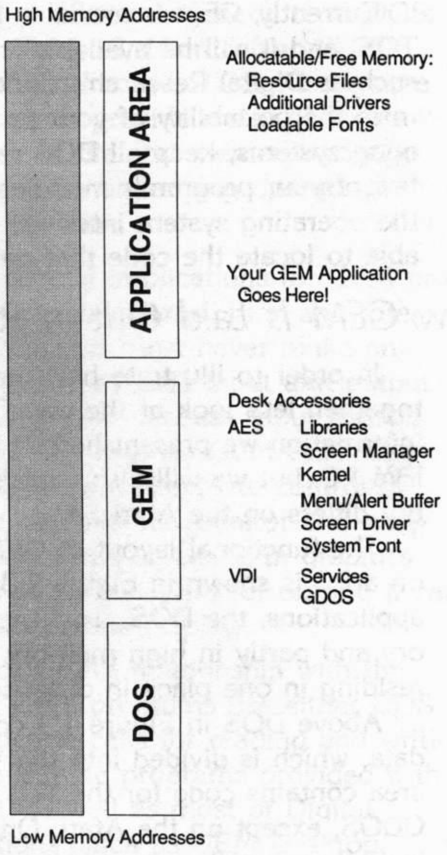


Figure 1.3: *The GEM Memory Map*

area when your GEM application gets loaded.

The menu/alert buffer is an area of RAM that is big enough to hold one-fourth of the screen. Whenever a drop-down menu or a form alert (a GEM warning or error message) is activated, the screen image beneath the menu or alert is saved into this buffer and is restored when the menu or alert is dispensed with. This is an optimization to speed up such common operations as the display of menus and warnings. GEM does not save the area under overlapped windows, but instead relies on the applications and accessories to redraw these areas when told to (we'll talk some more about this later in the chapter, in the section entitled "Division of Labor"). The amount of memory required for this menu/alert buffer depends on the resolution of the screen device. To understand how much memory might be required, see the section on "Raster Operations" in Chapter 3.

The size of the desk accessory area, like that of the menu/alert buffer, may also differ on various systems. GEM will load up to three desk accessories. Before each accessory is loaded, GEM checks to see if there is enough room to load the GEM application. Early versions of GEM reserve 128K for the application; the latest version of GEM reserves approximately 192K. GEM won't load any more desk accessories if there isn't enough memory left for the GEM application.

On the IBM PC, the early versions of GEM can load themselves into 256K and leave 128K for the GEM application (depending on the screen device, it may not be able to load many desk accessories, however). Because it is slightly larger and because it reserves more room for the application, version 2 of GEM requires 384K of RAM. The GEM application is responsible for managing the memory set aside for it. It is very important to remember, however, that the application must leave some memory unallocated in order to let the GEM functions that allocate memory work properly.

WHAT GOES INTO BUILDING A GEM APPLICATION

With this survey of the components of GEM in mind, let's look briefly at what goes into building a GEM application. We will talk about the components of the program, including the bindings, resources, and overall program structure. Then we will discuss the GEM Developer's Kit, along with what assistance you can expect from it in learning how to program and in actually programming GEM applications. Finally, we will describe the coding conventions for GEM and for this book, as well as what we've used for development systems.

The Components of a GEM Application

For purposes of this discussion, we can divide up the typical GEM application into the following components:

— Bindings
— Resources
— Application-specific code

Let's discuss each of these components in turn.

The Bindings

The first component, the *bindings,* basically consists of a set of procedures in the language that the program is written in. In our examples, the bindings are mostly in C, with a tiny amount in assembler code. These bindings, which are provided in the Developer's Kit, basically take their parameters and pack them into the GEM interface arrays, along with a function number to tell GEM which function to perform. Once the arrays have been filled, the bindings call a small assembler routine with the addresses of the arrays. This small assembler routine executes an interrupt on the IBM PC architecture or a software trap on the Atari ST architecture, and thus passes control to GEM. This kind of interface is almost exactly the method used for calling DOS in both the PC and the Atari environments. Once GEM has performed its assigned task, it returns to the binding function, and any information to be passed back to the application is unloaded from the arrays into the designated storage.

The Developer's Kit provides some additional code that you can think of as a small run-time library. This additional code includes start-up modules to make sure the application releases enough memory. It also includes some miscellaneous string handling and arithmetic functions and bindings to DOS functions.

The bindings are grouped together into several large files. You may want to unpack these binding files into smaller files in order to exclude the binding functions (and code) that your application never uses, thereby reducing the size of your application.

Resources and Resource Files

The second component of a GEM application, the *resources,* are data structures that are used to build display and/or input specification information for certain AES functions. Resources are used to represent the menu bar and all of its associated submenus, form alerts, and dialogs.

Although your program can build resources, they are usually complex enough to make it undesirable to do so. One of the nicest features of GEM is a tool called the *Resource Construction Set,* or *RCS,* which comes in the GEM Developer's Kit. The specific purpose of the RCS is to assist you in building resource files. Resource files, which contain the resource data structures, can be loaded separately by your application. Although this scheme may sound overly complicated, it has some very important advantages.

First, it allows you to construct the images, dialogs, and alerts that your application uses before you write any application code. Thus, the RCS can serve as a prototyping tool for your application.

Second, the resource file can be edited separately from your program. This means that any text to be displayed by your program can be changed without the need for modification to the program itself. Thus, somebody who speaks French can translate the resources for your application into French, and suddenly there's a whole new market for your program. Because it offers these capabilities, the Resource Construction Set is, in our opinion, one of the most exciting parts of GEM.

Application-Specific Code

The third component of your GEM application is the application-specific code that you develop to make your program do what it was designed to do. The structure of the GEM services has a couple of implications for the way you structure your program. We will preview these implications here and then discuss them in greater depth in Chapter 6.

Division of Labor

The first implication to be aware of is the *division of labor* associated with GEM programs. In order to run as efficiently as possible, GEM performs part of the job and expects your application to perform the rest. An example of this concept is GEM's approach to handling overlapping windows. Some windowing systems will save the window area when another window is overlaid on the current window. The designers of GEM felt that this approach required too much memory, and so every GEM application must be able to redraw the contents of a window whenever that window is uncovered. This requirement to redraw the screen at any time makes programs a bit more difficult to write.

There are several other examples of functions that require the cooperation of your program and GEM. For example, the AES Screen Manager handles a considerable amount of mouse activity in cooperation with your program, including any user selection of a menu item or click on a window control point. When the user wants to resize the window, for instance, the Screen Manager handles all of the user interaction from when the user clicks on the sizing box to when she releases the button to indicate the new window size. When the button is released, the Screen Manager sends your application a message with the new size. Your application can either reset the window size by calling GEM with the size or not do so, depending on whatever it determines is appropriate.

Event-Driven Code

Another way in which the structure of GEM services impacts the structure of examples in this book and of your own code has to do with *event-driven programming.* Event-driven programming is a style of building programs that makes for extremely interactive applications. GEM provides a function named **evnt_multi( )** to wait for one or more events at any time. Most programs use **evnt_multi( )** to accept user input. The ideal here is to handle any kind of input from the user in a consistent fashion, which can be more easily achieved if all of the input is handled from a single point in your program. This style of coding is conducive to avoiding program *modes,* or input states, which are difficult for the user to recognize and deal with.

We will discuss event-driven programming in more detail in Chapter 6, where we will also discuss various other design considerations. In addition, the examples throughout this book have been written with this basic principle of GEM program structure in mind.

The GEM Developer's Kit

The GEM Developer's Kit is a collection of tools, programs, and manuals that facilitate the construction of

GEM programs. Although it would be possible to construct bindings for GEM, there are several tools included with the Developer's Kit that make it good value.

What's in the Kit

The Digital Research GEM Developer's Kit includes

- C bindings for GEM AES and VDI
- The GEM Resource Construction Set
- The GEM Icon Editor
- GEMSID, a symbolic debugger
- Reference manuals for all of the above
- Examples of GEM applications (HELLO and DEMO)

We've already talked about the bindings and the Resource Construction Set. The Icon Editor allows you to construct bit images to form icons. GEMSID is derived from Digital Research's SID86 symbolic debugger, but it contains some modifications to assist the GEM programmer. The reference manuals are a bit terse, and Digital Research plans to release new documentation in the near future. There are many pieces of sample code included in the Kit, much of which is very informative when you take the time to study it carefully. The extent of the commenting in the sample code ranges from inadequate to fairly extensive.

Digital Research provides technical support for GEM programmers through the public network CompuServe. Part of the support is available via a public forum, but the most valuable part of the service, called GPS (for "GEM Programmer's Support"), is available as an added-cost subscription. The GPS members have access to more direct contact with technical service representatives, as well as to a large amount of code contributed by other GEM programmers.

What's Not in the Kit

The GEM Developer's Kit does not include, however, a number of tools vital to GEM programming. The Kit contains no compiler, assembler, or linker, even though it does contain bindings for a variety of languages. In Chapter 6 we present a brief critical survey of the tools we've used in writing GEM applications.

Development System Recommendations

You must have 384K RAM, a dual floppy system, and a graphics card in order to run a GEM application on an IBM PC or compatible computer. We recommend that you have a hard disk. In this book, we've used an IBM PC XT with 512K RAM, a 10MB hard disk, and an IBM Color Graphics Adapter card. We've also used an IBM PC AT with 1.5MB RAM and an Enhanced Graphics Adapter card.

We've tested all our examples on an Atari 520 ST with 1MB RAM and a 15MB hard disk, which we recommend for serious software development. We do not know if it is practical to develop GEM applications on an Atari system with a single floppy drive. An alternative for development of Atari GEM applications is to develop first on the IBM PC and then port the application over to the Atari. We recommend this method if you are planning to port your application, because it is easier to port from the Intel architecture to the Motorola architecture than it is to port the other way.

C Language Conventions

While the C language is a fairly portable language, we have found that there are some differences between compilers, operating systems, and microprocessor architectures that can be worked around by using certain programming conventions. There are two files included with the Developer's Kit that contain macro definitions (the C language #define) that can be changed for different environments to allow the code using these macros to work in a consistent manner across the different environments.

The first file, called PORTAB.H, contains a number of alternative type identifiers. In C, the integer type can be one size (16 bits, for example) on one machine, and another size (32 bits) on another. We use WORD (in capital letters, to emphasize that this is a macro) to be whatever type declaration the compiler needs to produce a signed, 16-bit integer. A list of the types we use includes

```
#define   BOOLEAN   short int        /* 16 bits */
#define   VOID      int              /* nothing */
#define   BYTE      signed char      /* 8 bits */
#define   UBYTE     unsigned char    /* 8 bits */
#define   WORD      signed short     /* 16 bits */
#define   UWORD     unsigned short   /* 16 bits */
#define   LONG      signed long      /* 32 bits */
```

The second file, MACHINE.H, helps to handle details like pointer size and byte ordering. In small model applications on the IBM PC, a pointer requires only 16 bits, but it can only address the program's local data area (within 64K). The GEM AES requires accessing information outside of this area (for example, in resource files). Therefore, the example programs often use the LONG type to hold pointers that can point anywhere in memory, and the ADDR() function to turn a short local pointer (16 bits) into a 32-bit global pointer. Since the 68000 architecture ordinarily uses 32-bit pointers, the ADDR() function just returns the pointer itself.

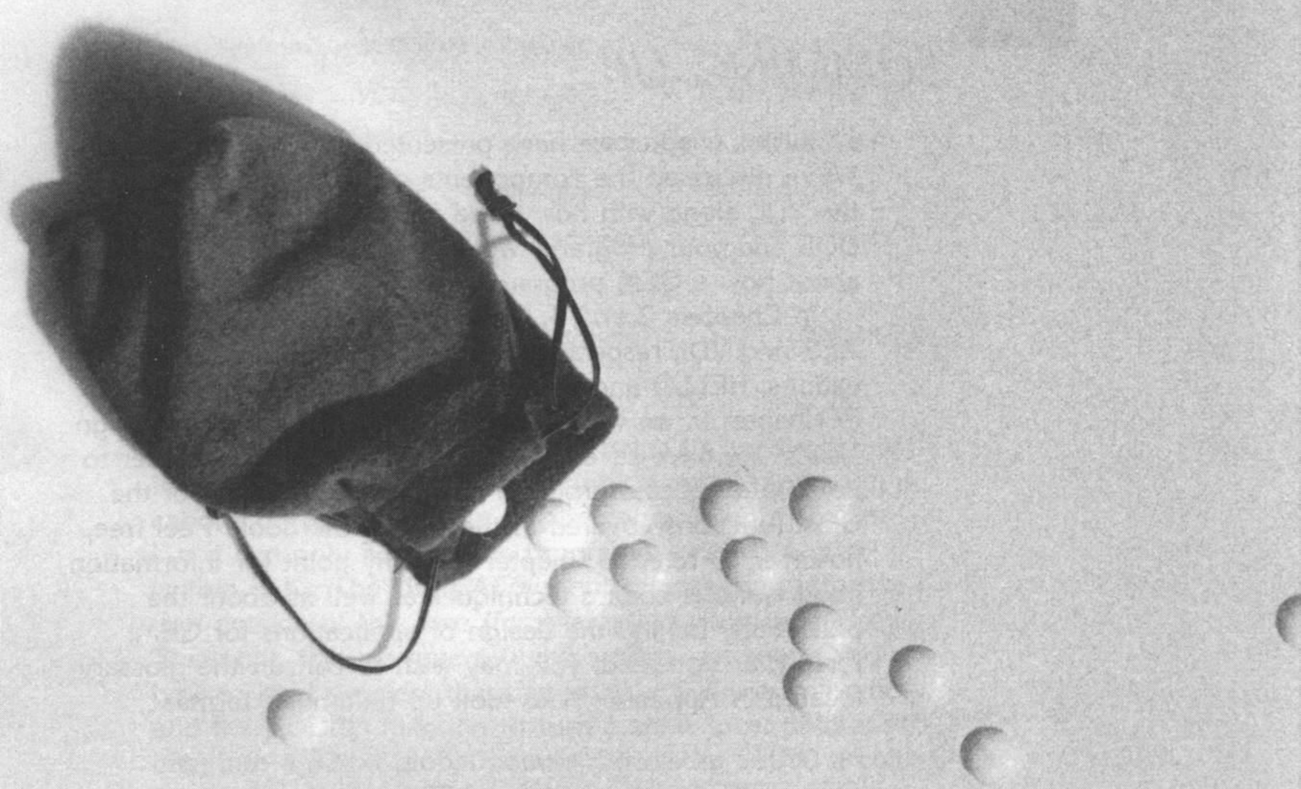
Some Conventions Used in This Book

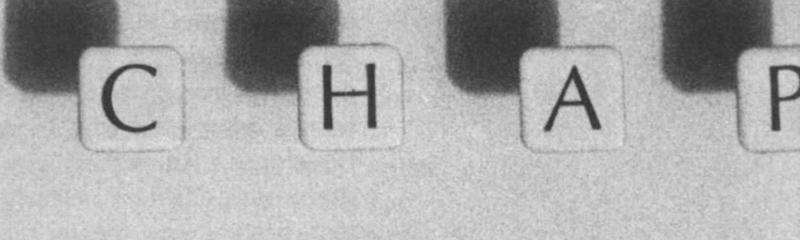
Please note that throughout this book, all GEM function will be printed in boldface and in a special **program font**. The same program font, minus the boldface, will be used for the examples presented in the text as well as for the variable names associated with those examples or with the GEM Developer's Kit. In this way, anything associated with programming GEM will clearly stand out from the rest of the text. Finally, any special terms associated with GEM will be italicized when we introduce and define them.

SUMMING UP

In this chapter we have presented an overview of GEM. We've discussed the components of GEM, the AES and the VDI, along with how these components work with DOS and your program. We've also talked a little bit about how a GEM program gets built.

In Chapters 2 and 3, we delve into the details of the AES and VDI, respectively. We explore two sample applications, HELLO and DEMO, in detail in Chapters 4 and 5. In Chapter 6, we will present a number of program design issues. We have saved this information for last in order to be able to discuss program design in the context of the GEM functions covered in the rest of the book. Feel free, however, to refer to Chapter 6 at any point for information about various coding techniques as well as about the philosophy behind the design of applications for GEM. Finally, as you read, you may wish to consult the glossary located in Appendix A to look up unfamiliar terms.




C H A P

T E R

2

APPLICATION ENVIRONMENT SERVICES

This chapter presents the features of the Application Environment Services (AES), which include event handling, objects, and window management. We will have a long talk about the GEM event-handling system. Perhaps the second most signficant topic in this chapter is the concept of a GEM object. Because the original DRI documentation does not present a clear picture of an object and its importance, we have greatly expanded on the DRI discussion. We will also introduce GEM menus, alert boxes, forms, dialogs, and windows.

In the following sections, we will be discussing each individual AES library. At the beginning of each section, we list the routines contained in the library. Not all of the functions will be discussed, but we will give a sample C language calling sequence for each library function at the end of each section.

We have attempted to illustrate the discussion with code samples. However, to provide small complete programs to illustrate each AES function (or group of functions) is impossible. We do provide small programs that illustrate the VDI calls (the next chapter), but because the AES builds larger, more complex graphical objects from the VDI primitives, we must use pieces of larger programs to illustrate the AES functions. As a way of doing this, we have taken to unabashed forward referencing to code in the DEMO program presented in Chapter 5.

OVERVIEW OF THE AES'S ROLE IN GEM

Basically, the AES provides a set of routines to the programmer so that he doesn't have to reinvent the wheel. The AES is like a template: a set of useful, partially filled in forms that make your programming job easier. For instance, you could build your own version of a window using the graphics tools available in the VDI. Chances are you would also build a set of routines that manipulate your windows. So you would build a window creation function, then a window movement function, then a window close, delete, and so forth. All these routines and more are made available to you through the AES. There are, however, some rules to follow, which we will explain as we deal with each library of the AES both here and in Chapters 4 and 5.

A major strength in using the AES is that it provides consistency in two ways. First, there is the consistency that comes from giving the user the same interface for many different applications. Second, there is the consistency that the programmer gains from using the same routines for different applications. Neither the user nor the programmer has to learn a different way of doing things.

THE COMPONENTS OF THE AES

As we mentioned in Chapter 1, the AES is composed of a limited multitasking kernel, a Screen Manager, and 11 libraries: Application, Event, Menu, Object, Form, Graphics, Scrap, File Selector, Window, Resource, and Shell. Figure 2.1 illustrates the organization of the AES. The kernel is simple and controls the various tasks of the GEM environment, which includes the Screen Manager. The Screen Manager is the controlling agency that handles all user input when the mouse cursor is off the window's work area. Both these modules will be discussed in greater detail in the next section.

All the functions of the AES that are available to the programmer are organized in sets of related functions called *libraries.* For instance, all the functions that control windows are grouped in the Window Library, all the event functions in the Event Lbrary, and so on.

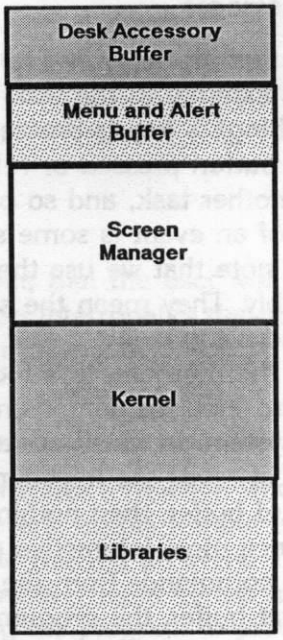


Figure 2.1: *The Organization of the AES*

GEM EVENTS

The GEM AES Event Library provides the foundation that governs all user input in a GEM application. First, let's define an *event* to be some action that is external to the current task and that may require attention. The AES defines the following events:

— Keyboard interrupts
— Mouse movement

— Mouse button changes
— Timer expiration
— Messages

Notice that all these events are either I/O events or actions external to the task (which is really saying the same thing): a key is pressed, the mouse is moved, a mouse button pressed or released, a message is received from another task, and so on. In any of these cases, the action of an event is some sort of input to the task. (Please note that we use the terms *task* and *process* interchangably. They mean the same thing in the context of this discussion.)

For the moment, let's focus on what happens during keyboard input to demonstrate the utility of the GEM event system. In most interactive applications, the program just reads the keyboard. If there is nothing in the keyboard buffer, then nothing is returned to the program. Now since most interactive programs depend on getting control commands from the user, when there is nothing in the input buffer, the program can do nothing and is forced to keep asking the user for input. Thus, the program continually polls the keyboard, a procedure which is called *busy waiting.*

The following loop is an example of polling the keyboard:

```
read keyboard
while(character not entered){
    read keyboard
}
```

Such a polling loop leaves the CPU unavailable for any other task. If you have many devices to handle, what happens if another event occurs while you are polling a different device? For instance, what if the application expects input from both a keyboard and a mouse? In this case, the polling loop changes to the following:

```
while( ){
    if (read keyboard) then do_keyboard( )
    else
    if (read mouse) then do_mouse
}
```

It is possible, however, that the user will move the mouse while the polling loop interrogates the keyboard, resulting in a missed mouse movement.

The Multiple Event Handler

The GEM AES provides a function whereby your program can wait for several events without either tying up the CPU by polling or missing an event. The kernel wakes your program up whenever input comes from the specified devices, thus allowing the program to process the data before going back to sleep to wait for the next set of input. This function is called the *multiple event handler,* **evnt_multi( )** (see Figure 2.2). **evnt_multi( )** keeps everyone happy: the system isn't slowed down by one process hogging the CPU, and each application can get the kind of information it needs. In the case where several events happen simultaneously, each event is buffered by GEM and made available to the application when it is ready to handle the event.

The ability of GEM to handle multiple events is a feature that distinguishes it from the Macintosh system. GEM's event system provides a more efficient mechanism to handle the many different inputs of a graphics environment. Because GEM was initially designed for a much slower and inefficient CPU and operating system, the Intel 8088 running MS-DOS, its event system had to be more efficient than the Macintosh's.

```
which = evnt_multi(MU_BUTTON | MU-MESAG | MU-MD1,
                   num_clicks, wich_button, but_state,
                   mouse_in_out, mevnt1_x, mevnt1_y,
                   mevnt1_width, mevnt1_height,
                   0,0,0,0,
                   addr_msgbuf,
                   lo_timint, hi_timint,
                   &mouse_x, &mouse_y, &mouse_but, 0,
                   &nbut_times)
```

where

INPUT:

num_clicks—# of times application expects a click.
wich_button—Which button was clicked.
but_state—Button state application is waiting for.
mouse_in_out—Waiting for entry/exit of rectangle.
mevnt1_x—X coordinate of mouse 1 rectangle.
mevnt1_y—Y coordinate of mouse 1 rectangle.
mevnt1_width—Width of mouse 1 rectangle.
mevnt1_height—Height of mouse 1 rectangle.
0,0,0,0—We are not using mouse 2 rectangle.
addr_msgbuf—Address of 16-byte message buffer.
lo_timint—Low word of timer interval (LONG).
hi_timint—High word of timer interval.
0—We are not waiting for keyboard event.

OUTPUT:

mouse_x—X coordinate of mouse when event occurred.
mouse_y—Y coordinate of mouse when event occurred.
mouse_but—Button state when event occurred.
nbut_times—# of times button entered state.

Figure 2.2: *Example of evnt_multi( ) Call*

evnt_multi( ) allows the application to wait for any and all types of events (keyboard input, mouse button changes, mouse movement, messages, or a timer running out).

evnt_multi() has the largest set of parameters of all the GEM functions, because it essentially duplicates the work done by all of the other event functions. Depending upon how many and what kind of events you want to wait for, **evnt_multi()** needs information for up to two sets of mouse events as well as for what events to wait for, where to store the return values in the case of a keyboard event, and so on.

evnt_multi() waits until one or more of the events that the programmer has specified occurs, and returns which events happened. For instance, look at Table 2.1 to see the value of the input flag that identifies which events to wait for. If the flag was set to 0x003F, for example, the caller would be waiting for all the possible events (the sum of all the defined flags in Table 2.1). If the flag were 0x0017, then **evnt_multi()** would wait for either a keyboard, mouse button, mouse movement, or message event. To find out which of the specified events happened, check the return value of the **evnt_multi()** call (the variable which in Figure 2.2). **evnt_multi()** returns the logical OR of all events that the user was interested in and that occurred. For instance, if the user used our first flag example (0x003F) and both a mouse movement event happened and a message happened, then the return code from

Flag	Defined Name	Event
0x0001	MU_KEYBD	Keyboard
0x0002	MU_BUTTON	Mouse Button
0x0004	MU_M1	Mouse 1 Movement
0x0008	MU_M2	Mouse 2 Movement
0x0010	MU_MESAG	Message
0x0020	MU_TIMER	Timer

Table 2.1: *Definition of the Event Flags*

evnt_multi( ) would be 0x0014 (0x0010 for the message event and 0x0004 for the mouse movement).

See Listing 5.6 in Chapter 5 for an example of using **evnt_multi**( ).

When the application calls **evnt_multi**( ), the application really no longer needs to use the CPU, since the application wants to wait until some I/O happens, which is not a CPU function. In this case, the application is said to be *blocked* from execution until some specific external event happens. For efficiency reasons, the AES kernel replaces the blocked application with a process that is ready to run, such as the Screen Manager.

The Kernel

The GEM kernel places a blocked process on the *Not Ready list,* which contains all the processes that are waiting for some kind of external event (usually I/O) that hasn't happened yet. Of course, there is also a *Ready list* containing all the processes that are ready to run—that is, not waiting for any outside event. By definition, the process on top of the Ready list is currently running. Figure 2.3 illustrates how the kernel dispatches processes.

The kernel's job is to move the processes in and between the two lists. The Ready list is managed in what is known as a *round robin scheme without preemption.* "Round robin" means that when a process comes off the top of the Ready list and is either moved to the Not Ready list or merely terminated, the remaining processes on the Ready list are moved up one position, as shown in Figure 2.3. "Without preemption" means that when a process is added to the Ready list, it is always added to the bottom of the list, and no task has priority over other tasks. Each gets executed in the order in which they appear in the Ready list.

A process is said to be dispatched when it is moved from the Ready list to the Not Ready list. Thus, you might hear of the term *dispatcher* in discussions of the kernel's operation (a dispatcher or *scheduler* is part of a kernel).

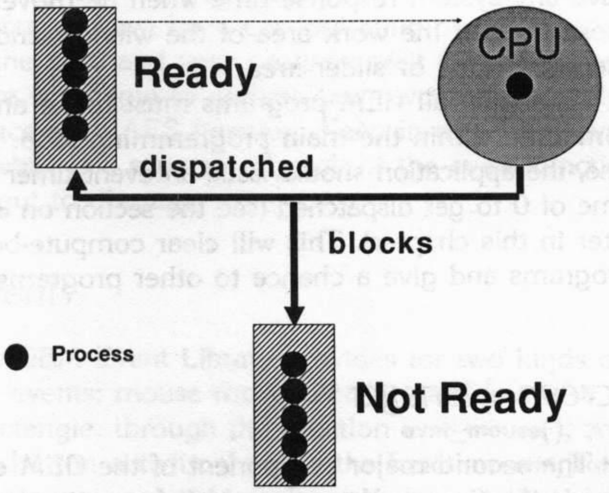


Figure 2.3: *How the Kernel Dispatches Processes*

In contrast to the Ready list, the ordering of the Not Ready list has no bearing on how tasks are removed from it. The dispatcher takes all the tasks that are ready to run off the Not Ready list and adds them to the Ready list in the order that the tasks become ready.

As we have said, the GEM kernel is a limited multitasking system in that it can only handle five tasks: three desk accessory programs (containing up to six desk accessories), one application, and the Screen Manager.

It is important to note that the only time that GEM performs a dispatch is when a GEM AES call is made. It is possible for the application to use only a few AES calls at the initialization stage (like those for creating and opening a window), and then use only VDI functions in the rest of the code. The kernel is an AES component, and if the AES is not used, no multitasking will take place; in particular, the Screen Manager will not run. If the Screen Manager does not get to run, in turn, the user will not have any system response time when he moves the mouse out of the work area of the window and onto the menu bar area or slider area.

As a rule, all GEM programs must make an AES call sometime within the main programming loop. If nothing else, the application should issue an event timer call with a time of 0 to get dispatched (see the section on **evnt_timer**( ) later in this chapter). This will clear compute-bound GEM programs and give a chance to other programs.

The Screen Manager

The second major component of the GEM event system is the Screen Manager, which keeps track of the mouse when it is outside of the work area of the active window. The application must track the mouse when it is within its own work area, but only when there is a change of state (that is, when mouse button is pressed or mouse moves into or outside of a rectangle). The areas in question are the borders of a window, the drop-down menus, and the menu bar. This means when the mouse crosses the area of the screen where the drop-down menus reside, the Screen Manager sends the appropriate message.

The Screen Manager is constantly tracking where the mouse is. As soon as the mouse crosses into the menu bar area, the Screen Manager saves the screen area where the appropriate menu is going to be displayed, and then displays the drop-down menu onto the screen. The Screen Manager then waits for some selection on the part of the user—that is, clicking on a menu item or other part of the screen. If the mouse button is pressed and released on a menu item, then the Screen Manager sends a message to the running application with the information as to which menu item was selected. If the button is pressed off the menu area, then the Screen Manager restores the original screen area from the menu buffer.

So far we have been concentrating on what happens when the AES and your program wait for many events. Now we will begin to discuss each event individually, and introduce each AES function. Keep in mind, however, that **evnt_multi**( ) is a superset of each of the event functions we are about to discuss.

Mouse Events

The GEM Event Library provides for two kinds of mouse events: mouse movement into and out of a specified rectangle, through the function **evnt_mouse**( ), and mouse button activity, through the function **evnt_button**( ). Of course, any pointing device may be substituted for a mouse as long as it is functionally equal to a mouse. Thus, a sketch pad, data tablet, or light pen may be viewed through the GEM system as a mouse.

First let's deal with mouse movement. The typical GEM window is divided into two major areas: the window *work area,* which is managed by the application itself, and the rest of the window (title bar, menu bar, and so on), which is managed by the Screen Manager. When the mouse moves, it may do so in relation to a *rectangle.* For instance, the mouse may move anywhere inside a rectangle, and there will be no particular meaning associated with such movement. However, if it moves out of that rectangle, then the application may want to know about it and begin a series of actions in response. This is known as a *mouse movement event.* In the simple case, the rectangle may be identified as the cursor's current position.

The **evnt_mouse**( ) function allows the user to identify

HOW A GEM RECTANGLE IS SPECIFIED

A rectangle in GEM AES is specified by a structure with an (x,y) coordinate and a width and height component. For instance, a 10×10 rectangle can be placed anywhere in the window by specifying the starting x and y coordinate and a width of 10 and height of 10 (or (x,y,10,10)). In this system, (0,0) is at the upper left corner of the window. (Note, however, that the y coordinate does not go negative; rather, it remains a positive number.) Thus, (0,4) means 4 units below the origin on the y axis. If we wanted a 10×10 rectangle at that point, we would specify a rectangle structure of (0,4,10,10), which results in a rectangle at points (0,4), (10,4), (0,14), and (10,14).

In the code examples in Chapters 4 and 5, you will see a C structure called GRECT *used. It defines an x coordinate, a y coordinate, a width, and a height. To specify a rectangle for the VDI, however, a different scheme is used: two opposing corners are specified, which is conceptually the same as specifying one point, and a width and height. To accommodate the difference in the systems used by the AES and the VDI, there are functions to translate between the two (***GRECT_to_array( )***, for example).*

whether the mouse movement was into a specified rectangle or whether it was out of a specified rectangle. This allows the user to change the *mouse cursor form* or just *mouse form* in accordance with some meaning associated with the specified area of the screen. You can observe an example of this change in mouse form when you move the mouse over the menu bar in the GEM DEMO program presented in Chapter 5. (See also the discussion of the **graf_mouse( )** function in the section on the Graphics Library and Listing 5.7 in Chapter 5.)

In addition to keeping track of the mouse movement, it is important to monitor changes in the *mouse button state* by means of **evnt_button( )**. For instance, what if the user presses the third button on the mouse (assuming that the mouse has a third button)? If all you care about is the first button, the application can tell **evnt_button( )** to ignore this action. In fact, you can specify which button you are interested in as well as whether the application is waiting for it to be pressed or released. See Listing 5.8 in Chapter 5 for an example of handling button activity.

Another event function associated with the mouse, **evnt_dclick( )**, is the length of time to wait for a *double-click.* In most graphics user interfaces, the user may press upon the same mouse button twice in quick succession (this is double-clicking) in order to signify selection and action. Thus, double-clicking in many instances provides a shortcut to the usual point-and- select-and-point-and-select operation of execution by including an embedded action that is usually execution. For instance, in the GEM Desktop the user may click on an executable folder (like GEM Draw), move the mouse to the menu bar, select the File menu, and open the selected folder (in this case, since the folder identifies an executable file, GEM will run that program). However, a savvy user will instead merely double-click on the folder. If this has meaning, then GEM will run the selected program. If the folder cannot be opened or executed, then nothing will happen.

The amount of time that the program will spend waiting for the second click certainly has great influence on the actions of the user. A very short time will mean that the user must be very fast to get the second click in. If the click arrives outside the click window, the second click is interpreted as the first click of the next click pair. If the click interval is very long, on the other hand, the mouse driver waits a long time before reporting a completed click event. No second click will be misinterpreted, but there is a lot of wasted time at the mouse driver level.

In the GEM Desktop, the double-click interval can be changed by the user, and this setting is carried to all applications run from the Desktop. If your application does not load from the Desktop (that is, if it loads directly), the default dclick setting is 2, an intermediate speed. As you can control the dclick setting by means of **evnt_dclick( )**, you can write your application to allow the user to change the dclick setting so that it is more comfortable for him.

Keyboard Events

While GEM provides a graphics interface to the computer, this doesn't obviate the need for input from the keyboard. In fact, GEM allows for a mouseless system that will accept cursor movement control from the cursor control keys (the arrow keys on the keypad). In this case, however, the keyboard driver is responsible for tracking the cursor, not the application. The application should therefore never receive an interrupt concerning the cursor movement. Yet if the application expects keyboard input, it must handle this input using the **evnt_keybd( )** function. The state of the keyboard is also available through an additional AES call, **graf_mkstate( )**.

None of the code samples included in this book deal with using the keyboard. You might look at the GEM DEMO application distributed by DRI in the Developer's Kit for an example of using the keyboard. (Please do not confuse the DEMO program included in this book with the one distributed by DRI.)

Message Events

In general, all communication within the GEM system is driven through the message system. While the user may move the mouse all over the screen, the Screen Manager only sends a message to tell waiting applications significant results: for example, selection of a menu item, the need to redraw the screen, slider bar movement, and so on. The GEM message system uses fixed-length messages through a pipe, and these messages are *FIFO* (first in, first out) ordered.

(A *pipe* is an FIFO entity that acts exactly as you would expect: information that goes into a pipe comes out first, just like water flowing through a water pipe.) Messages are taken out of the pipe when the application reads the pipe.

GEM provides 12 predefined messages and allows more to be defined by the application. Each predefined message is 16 bytes long and follows a set format:

Word 0:	Message type.
Word 1:	Application ID of sender.
Word 2:	Number of extra bytes (greater than 16). In order to obtain the rest of the message, use the AES **appl_read()** routine.
Word 3–7:	Depends on the message.

The 12 predefined messages are

MN_SELECTED —A menu was selected.

Word 0:	10
Word 3:	Object index of the selected menu title.
Word 4:	Object index of the selected menu item.

WM_REDRAW —Redraw the screen.

Word 0:	20.
Word 3:	Handle of window to redraw.
Word 4:	X coordinate of the area to redraw.
Word 5:	Y coordinate of the area to redraw.
Word 6:	Width of the area to redraw.
Word 7:	Height of the area to redraw.

WM_TOPPED —Make this window the topmost.

Word 0:	21.
Word 3:	Window handle.

WM_FULLED —Clicked upon the full box.

Word 0:	23.
Word 3:	Window handle.

WM_ARROWED —Clicked on the arrows or scroll bars.

Word 0:	24.
Word 3:	Window handle.
Word 4:	The requested action:
	0—Page up
	1—Page down
	2—Row up
	3—Row down
	4—Page left
	5—Page right
	6—Column left
	7—Column right.

WM_HSLID —New horizontal slider position.

Word 0:	25.
Word 3:	Window handle.
Word 4:	0–1000, indicating the slider position, where
	0 = Leftmost position
	1000 = Rightmost position.

WM_VSLID —New vertical slider position.

Word 0:	26.
Word 3:	Window handle.
Word 4:	0–1000 (see WM_HSLID).

WM_SIZED —The new window size.

Word 0:	27.
Word 3:	Window handle.

	Word 4:	New x coordinate.
	Word 5:	New y coordinate.
	Word 6:	New width.
	Word 7:	New height.
WM_MOVED		—The new window coordinates.
	Word 0:	28.
	Word 3:	Window handle.
	Word 4:	New x coordinate.
	Word 5:	New y coordinate.
	Word 6:	New width.
	Word 7:	New height.
AC_OPEN		—Desk accessory item selection.
	Word 0:	30.
	Word 3:	Desk accessory menu item identifier.
AC_CLOSE		—Close any desk accessory selections.
	Word 0:	31.
	Word 1:	Desk accessory item menu identifier from **menu_register( )** call.

The **evnt_mesag( )** function is used to wait for any expected messages. (If you don't want to receive any messages, then take the receiver off the hook by not using **evnt_mesag( )**.) In the case of the 12 predefined messages, the entire message is contained in the 16 bytes. Even if you define your own messages, it is certainly possible to deliver the entire message within the 16-byte format. For times when you need to transfer more than 16 bytes of information, however, you must use two functions from the Application Library: **appl_read( )** and **appl_write( )**.

appl_write( ) writes a specified number of bytes into a message pipe between two processes. **appl_read( )** takes a specified number of bytes out of the pipe. Thus,

evnt_mesag( ) may be used to transfer the actual number of bytes in the write message, before issuing an **appl_read**( ). In any case, the point to remember is that all the message events only pass 16 bytes of data. You must use the **appl_read**( ) routine to get the rest.

See Listings 5.15, 5.16, 5.17 in Chapter 5 for a complete treatment of handling messages.

Timer Events

GEM provides a function called **evnt_timer**( ) that allows the application to wait a specified amount of time without doing anything as silly as polling the system clock. The application just goes to sleep for the requested amount of time and is then awakened with a timer event.

The event is guaranteed to happen within the specified time interval. It is possible, however, to request such a short interval that it takes longer for the dispatch, which means that you can't be certain of the accuracy of the *time out.* Similarly, if there are other processes running, the timed out process might take longer than requested before being awakened.

Event Library Syntax Summary

This section is a short reference summary of all the functions in the Event Library.

Waiting for a Button Event: evnt_button( )

The **evnt_button**( ) function waits for the specified button state. To use it, see the following syntax summary.

```
WORD nbut_times = evnt_button (num_clicks, wich_button,
                               but_state, &mouse_x,
                               &mouse_y, &mouse_but,
                               &kb_state)
```

Input:

WORD num_clicks	Number of clicks on the specified button that the application will wait for.
UWORD wich_button	Which button to wait for (up to 16).
UWORD but_state	Button state to wait for (up/down).

Output:

WORD nbut_times	Number of times the specified button entered the desired state.
WORD mouse_x	X coordinate of mouse when button event happened.
WORD mouse_y	Y coordinate.
UWORD mouse_but	Button state when event happened.
UWORD kb_state	The state of the keyboard Shift, Ctrl, and Alt keys when the button event happened.

Key	**Down Value**
Right Shift	0x0001
Left Shift	0x0002
Ctrl	0x0004
Alt	0x0008

A 0 value indicates a key up.

Getting and Setting the Double-Click Interval: evnt_dclick( )

The **evnt_dclick**( ) function gets or sets the interval between clicks that constitute a double-click. To use it, see the following syntax summary.

WORD speed = **evnt_dclick** (new, getset);

Input:

WORD new	New double-click speed. Slow to fast (0-2-3-4).
WORD getset	Get current value (0) or set new value (1).

Output:

WORD speed	The actual double-click speed (0-4).

Waiting for a Keyboard Event: evnt_keybd()

The **evnt_keybd()** function reads the keyboard and returns the key that was pressed. To use it, see the following syntax summary.

UWORD key = **evnt_keybd( );**

Output:

UWORD key	The code of the key that was pressed (see the DRI documentation or the IBM technical manual for the IBM keyboard scan codes).

Waiting for a Message Event: evnt_mesag()

The **evnt_mesag()** function waits for a message. To use it, see the following syntax summary.

WORD dummy = **evnt_mesag** (msgbuff);

Input:

LPTR msgbuff	Address of a 16-byte message buffer in which the message will be placed.

Output:

WORD dummy	Always 1.

Waiting for a Mouse Event: evnt_mouse( )

The **evnt_mouse**( ) function waits for the mouse to enter or leave the specified rectangle. To use it, see the following syntax summary.

```
WORD dummy = evnt_mouse (mouse_in_out, mevnt_x,
                         mevnt_y, mevnt_width,
                         mevnt_height, &mouse_x,
                         &mouse_y, &mouse_but,
                         &kbstate);
```

Input:

WORD mouse_in_out	Do we wait for entry into (0) or exit from (1) the specified rectangle?
WORD mevnt_x	X coordinate of the rectangle that the mouse may enter or leave.
WORD mevnt_y	Y coordinate.
WORD mevnt_width	Width of rectangle.
WORD mevnt_height	Height of rectangle.

Output:

WORD dummy	Always 1.
WORD mouse_x	X coordinate of mouse when event happened.
WORD mouse_y	Y coordinate.
WORD mouse_but	Button state when event happened.
WORD kbstate	Keyboard state when event happened (see **evnt_button**( )).

Waiting for Multiple Events: evnt_multi( )

The **evnt_multi**( ) function waits for one or more of the specified events. To use it, see the following syntax summary.

```
WORD which = evnt_multi (events, num_clicks, wich_button,
                         but_state, mouse1_in_out,
                         mevnt1_x, mevnt1_y,
                         mevnt1_width, mevnt1_height,
                         mouse2_in_out, mevnt2_x,
                         mevnt2_y, mevnt2_width,
                         mevnt2_height, msgbuff,
                         lo_timint, hi_timint, &mouse_x,
                         &mouse_y, &mouse_but,
                         &kb_state, &key, &nbut_times);
```

Input:

UWORD events	Logical OR of which events to wait for (see Table 2.1).
WORD num_clicks	Number of times application expects a click.
WORD wich_button	Which button to wait for.
UWORD but_state	What button state to wait for (up/down—0/1).
UWORD mouse1_in_out	Waiting for entry (0) or exit (1) for first mouse rectangle.
WORD mevnt1_x	X coordinate of the first mouse rectangle that the mouse may enter or leave.
WORD mevnt1_y	Y coordinate of first mouse rectangle.
WORD mevnt1_width	Width of first rectangle.
WORD mevnt1_height	Height of first rectangle.
UWORD mouse2_in_out	Wait for entry or exit for second mouse rectangle.
WORD mevnt2_x	X coordinate of the second mouse rectangle that the mouse may enter or leave.
WORD mevnt2_y	Y coordinate of second rectangle.
WORD mevnt2_width	Width of second rectangle.

WORD mevnt2_height	Height of second rectangle.
LPTR msgbuff	Address of 16-byte buffer to place any messages.
UWORD lo_timint	Low word of the time interval (LONG value) to wait.
UWORD hi_timint	High word of time interval.

Output:

WORD mouse_x	X coordinate of mouse when event happened.
WORD mouse_y	Y coordinate.
WORD mouse_but	Button state when event happened.
UWORD key	The code of the key that was pressed.
UWORD nbut_times	Number of times that the mouse button entered the specified button state.

Waiting for a Timer Event: evnt_timer( )

The **evnt_timer**( ) function waits for the specified time (in milliseconds). To use it, see the following syntax summary.

WORD dummy = **evnt_timer** (UWORD ev_tlocount, UWORD ev_thicount);

Input:

UWORD lo_timint	Low word of timer interval (LONG).
UWORD hi_timint	High word.

Output:

WORD dummy	Always 1.

THE WINDOW LIBRARY

At the very heart of GEM lies the concept of windows. Simply speaking, a *window* is merely an area of the screen, but GEM enhances that area to include a set of borders and structures such as the title area, full box, and so on. A full-blown GEM window includes the following components, which are also shown in Figure 2.4:

— Title bar: to name the window.

— Move bar: to allow the user to drag the window around on the screen (occupies the same space as the title bar).

— Close box: to allow the user to close the window.

— Full box: to allow the user to toggle between the full and new sizes.

— Information line: to show application-specific information.

— Up/down arrows: to allow paging up or down one window unit.

— Scroll bar: to allow paging through a window.

— Left/right arrows: to move the columns left or right.

— Work area: the part of the window that does not consist of the above items and that is available for use by the application.

Perhaps at this point we should remind you not to begin to think of windows as something that contains anything. From the programmer's point of view, windows serve instead to organize the screen into handy units. It is up to you to keep things within the borders of the window

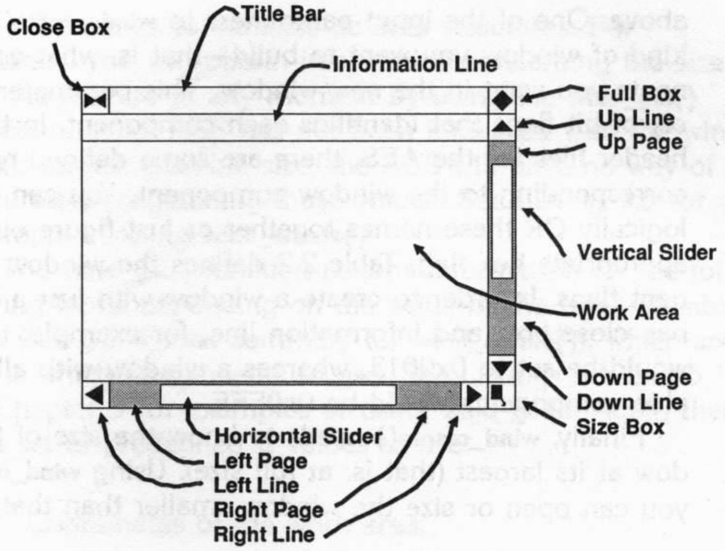


Figure 2.4: *GEM Window with All Components*

that you have created and put somewhere on the screen. It is your responsibility to keep track of where the window is as well as of drawing and clipping to that area. Your application can draw anywhere on the screen, which makes you responsible for confining your program's output to the window.

The value of the formal structure of windows lies in the components of the window that allow you to manipulate the data displayed within the borders of the windows. GEM provides you with a well-defined structure so you don't have to reinvent the wheel.

Window Creation and Sizes

By using the **wind_create**( ) function, you can choose to build a window with any of the components we have listed above. One of the input parameters to **wind_create()** is the kind of window you want to build—that is, what components you want in the new window. This parameter is a set of bit flags that identifies each component. In the header files for the AES, there are some defined names corresponding to the window component. You can either logically OR these names together or just figure out the appropriate hex flag. Table 2.2 defines the window component flags. In order to create a window with just a title bar, close box, and information line, for example, the flag would be set to 0x0013, whereas a window with all available components would be 0x0FFF.

Finally, **wind_create()** needs to know the size of the window at its largest (that is, at full size). Using **wind_open()**, you can open or size the window smaller than that, but

Flag	Defined Name	Window Component
0x0001	NAME	Title bar
0x0002	CLOSE	Close box
0x0004	FULL	Full box
0x0008	MOVE	Move bar
0x0010	INFO	Information line
0x0020	SIZE	Size box
0x0040	UPARROW	Up arrow
0x0080	DNARROW	Down arrow
0x0100	VSLIDE	Vertical slider
0x0200	LFARROW	Left arrow
0x0400	RTARROW	Right arrow
0x0800	HSLIDE	Horizontal slider

Table 2.2: *Definition of the Window Component Flag*

the AES needs to initialize its data structures. For instance, you can obtain information concerning the size for the window at any moment by using the **wind_get()** function. If the AES has not been initialized with the window's largest possible size, the AES will have no way of accurately maintaining those values. See Listing 5.5 for an example of using **wind_create()**.

The **wind_get()** function returns information on the following items depending on the value of the input variable kind (see the syntax summary for **wind_create()**), which specifies which components to use. See Listings 5.5 and 5.13 in Chapter 5 for examples of using **wind_get()**. Again there is a set of predefined C values to use:

— Coordinates of the work area.

— Coordinates of the current window, including borders, title bar, and information line.

— Coordinates of the previous window, including borders, title bar, and information line.

— Coordinates of the window at its greatest possible size, including borders, title bar, and information line.

— Slider position and size.

— Window handle of active window.

— Coordinates of the rectangles in the window's rectangle list.

— Where and how large the internal menu/alert buffers are.

As this list suggests, **wind_get()** is an important window function because it provides information that is critical, for example, in updating the screen (window rectangles), in resizing windows, and in moving data within a window based upon slider movement. In addition, **wind_get()** has a complementary function, **wind_set( )**, that allows you to change some of the fields handled by **wind_get( )**. For instance, when you change the current window coordinates, the AES will automatically change the previous window coordinate field and the current work area coordinates. You will not, however, be able to change all the **wind_get** fields directly. Listing 5.13 in Chapter 5 also shows a usage of **wind_set( )**.

Window Management

GEM uses the message system to communicate to the application when something has happened in the window. If the action occurred on the borders of the window, however, the Screen Manager processes the user's actions and sends the application a message telling it whether the user clicked on the full box, selected a menu item, dragged the window, resized it, or whatever. It is up to the application to process that information. For instance, based on the received message, the application can move a window or ignore the message. If you don't want a user to move a window, however, you shouldn't provide a move bar.

Because GEM provides overlapping windows, there is a formal, controlled way of updating and redrawing windows. The first rule is that the application takes the responsibility to control its window environment. The AES does not allocate memory for the application's window (or windows), nor does it update the screen when something happens to it. The AES does provide a consistent interface for the management of windows as well as services to the border area. The AES shares responsibility for the window display with the application, in that the AES tells the application where to draw and the application does the drawing.

When windows overlap, GEM must be able to tell which window is affected by a user action such as clicking with the mouse. For instance, if the user clicks with the mouse, either of two scenarios can occur.

First, if the window wherein the clicking occurred is the currently active window (called the *topmost window* or the *window on top*), the AES sends a message to the application if it is waiting for a mouse button event. Otherwise, the AES does nothing with the button event.

Let's use the GEM Desktop as a typical example of a GEM application and look at the highlighting it does when a user clicks on an icon. The Desktop waits for a mouse event, and receives a mouse button event wake up call with the coordinates of the mouse when the click takes place. The Desktop then uses **wind_find( )** (passing in the mouse coordinates) to find out which window the click occurred in. The Desktop arranges all the icons in a window as an object tree and keeps track of what tree occurs in what window. It is then able to do an **objc_find( )** to determine if the coordinates match an icon in the window. If the mouse was over an icon when the click occurred, then the Desktop highlights the icon. If the user clicked on the space between icons, the Desktop removes the highlighting from any highlighted icon.

Second, if the window where the click occurred is not on top, then the AES will send a message to the application that owns the window, telling the application that it must reorder its windows and that another window is now active.

Unfortunately, we cannot use the Desktop as an example here because of the changes made to it recently (it no longer supports overlapping windows). We can, however, examine the action of a desk accessory on top of a Desktop window. If you bring up the Clock desk accessory, the window that it uses will be on top of one of the Desktop windows. If you click on the underlying Desktop window, then the Desktop gets a message that it is now on top, which causes it to redraw its window so that the Clock is now hidden and the Desktop window is back on top.

Window Updating

Because GEM provides a small multitasking kernel and supports overlapping windows, the screen must be protected from inadvertent updating. Thus, GEM provides a mechanism whereby the application can keep other processes from updating the screen, ensuring that the application has a stable screen environment during the course of its modifications. This notion is similar to the problem of simultaneous update of a disk file in a concurrent system, and it is solved in a like manner—that is, by locking out any other process while the update is going on.

As you will see in the code samples in Chapters 4 and 5, **wind_update()** is the function that notifies the AES that the calling process wants to update the window. The AES then prevents any other process, including menus and alerts, from modifying that portion of the screen where the window resides (in the case of a full-screen window, the entire screen is locked). When the application is finished with the modifications, it calls **wind_update()** again to unlock the window screen area. If you forget to complete the update pair, you will notice that your menus will not work, nor your desk accessories, nor anything that changes the screen.

Window Redrawing

There are two reasons for a window to be updated: first, the user affects the window through, for example, scrolling, sizing, closing, or moving; and second, the application changes the displayed information. In all cases, either the specified area of the window or the entire window will need to be updated.

The mechanism that GEM employs is called "walking the rectangle list." For each window on the screen, the AES maintains a list of rectangles that defines the actual physical screen used by the visible part of each window. The rectangle list is generated by an algorithm that ensures that the screen is divided into the least number of nonoverlapping rectangles. For instance, if there is only one window on the screen and it is a full-screen sized window, then the list identifies only one rectangle, the window itself. Figure 2.5, on the other hand, shows an example of a two-window screen that requires six nonoverlapping rectangles.

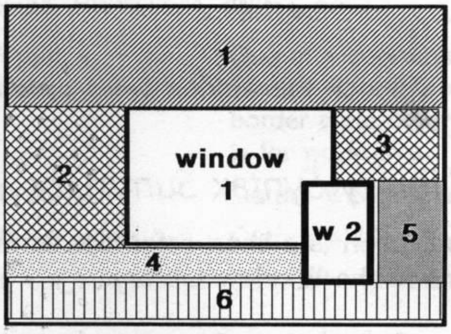


Figure 2.5: *Rectangle List*

When an application needs to redraw a window area, it first locks the window using **wind_update( )**. Then, using **wind_get( )**, it finds the first rectangle in the list. If this rectangle and the rectangle that defines the region that needs to be updated (called the *update rectangle*) intersect, then the application must compute the resulting rectangle and redraw that window area. The application gets the next rectangle in the list and continues using the technique we have just described until all rectangles in the window's list have been examined.

The following pseudocode shows the proper procedure. For an example of C code, look at Listing 5.14 in Chapter 5.

```
— 1  Lock down the window using wind_update(1).
— 2  Get the first rectangle in list:
     wind_get(WF_ FIRSTXYWH,
     &box(x,y,width,height))
— 3  While (box.width && box.height) {
     if (dest = intersect(redraw area, box)) {
         copy screen buffer to destination
     }
     get the next rectangle in the list
         wind_get(WF_NEXTXYWH, &box(x,y,width,height))
     {
— 4  Unlock the window (wind_update(0))
```

Window Library Syntax Summary

This section is a short reference summary of all the functions in the Window Library.

The Window Calculation Function: wind_calc( )

Given a window's work area coordinates, **wind_calc( )** calculates the borders, or given the border area coordinates, it calculates the work area. To use it, see the following syntax summary.

```
WORD ret_code = wind_calc (type, kind, in_x, in_y,
                          in_width, in_height, out_x,
                          out_y, out_width, out_height)
```

Input:

WORD type	Type of calculation: 0—return border area 1—return work area.
UWORD kind	Kind of window identified by which components it contains (see Table 2.2).

WORD in_x	Input x coordinate. If type = 0, then in_x is for the work area. If type = 1, then in_x is for the border area.
WORD in_y	Input y coordinate for type.
WORD in_width	Input width for type.
WORD in_height	Input height for type.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD out_x	Output x coordinate. If type = 0, then out_x is for border area; otherwise, out_x is for work area.
WORD out_y	Output y coordinate for type.
WORD out_width	Output width for type.
WORD out_height	Output height for type.

Closing a Window: wind_close( )

The **wind_close**( ) function closes an open window, preserving the handle and all allocated resources. The window may be re-opened. To use it, see the following syntax summary.

WORD ret_code = **wind_close** (handle);

Input:

WORD handle	Window handle of the window to close.

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.

Creating a Window: wind_create( )

The **wind_create( )** function creates the window, returns the window handle, and allocates the resources needed by the window. To use it, see the following syntax summary.

```
WORD ret_code = wind_create (kind, full_x, full_y,
                             full_width, full_height)
```

Input:

UWORD kind	Kind of window identified by which components it contains (see Table 2.2).
WORD full_x	X coordinate of the full-size window.
WORD full_y	Y coordinate.
WORD full_width	Width of full-size window.
WORD full_height	Height of full-size window.

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.

Deleting a Window: wind_delete( )

The **wind_delete( )** function deletes a window and frees all the resources attached to the window. To use it, see the following syntax summary.

```
WORD ret_code = wind_delete (handle);
```

Input:

WORD handle	Handle or identifier of a window.

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.

What Window Is the Mouse In: wind_find( )

The **wind_find**( ) function determines which window the mouse is currently over. To use it, see the following syntax summary.

WORD wat_wind = **wind_find** (mouse_x, mouse_y);

Input:

WORD mouse_x	X coordinate of the current mouse position.
WORD mouse_y	Y coordinate.

Output:

WORD wat_wind	The window handle of the window that the mouse is over.

Getting Window Information: wind_get( )

The **wind_get**( ) function returns either the GRECT values for various windows (current, previous, fullest, and so on), the slider position, the handle of active window, the GRECT values of first and next rectangle in the window's rectangle list, the slider size, or the location and length of the internal menu/alert buffer. To use it, see the following syntax summary.

WORD ret_code = **wind_get** (handle, flag, values[0]);

Input:

WORD handle	Handle of window.	
WORD flag	What information is requested:	
	1 Reserved	
	4 WF_WXYWH	Work area (x,y, width,height).
	5 WF_CXYWH	Current window, including borders,

	title bar, and information line (x,y, width,height).
6 WF_PXYWH	Previous window, including borders, title bar, and information line (x,y, width,height).
7 WF_FXYWH	Window at its greatest size, including borders, title bar, and information line (x,y,width,height).
8 WF_HSLIDE	values[0] = relative position of horizontal slider (1 to 1000); 1 is leftmost position, 1000 is rightmost.
9 WF_VSLIDE	values[0] = relative position of vertical slider (1–1000).
10 WF_TOP	values[0] = handle of topmost window.
11 WF_FIRSTXYWH	Coordinates of first rectangle in rectangle list (x,y, width,height).
12 WF_NEXTXYWH	Coordinates of next rectangle in rectangle list (x,y, width,height).
13 Reserved	
15 WF_HSLSIZE	values[0] = size of the horizontal slider relative to the scroll bar. −1 = the

	default, while 1 = smallest and 1000 = largest.
16	WF_VSLSIZE values[0] = size of vertical slider as above.
17 WF_SCREEN	Address and length of the internal menu/alert buffers.
	values[0] = low word of address.
	values[1] = high word.
	values[2] = low word of length.
	values[3] = high word.

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.
WORD values[4]	Return values depending on flag.

Opening a Window: wind_open( )

The **wind_open**( ) function opens the indicated window with the specified size and at the specified location. To use it, see the following syntax summary.

```
WORD ret_code = wind_open (handle, init_x, init_y,
                           init_width, init_height);
```

Input:

WORD handle	Window handle.
WORD init_x	X coordinate of window at the initial size.
WORD init_y	Y coordinate.

WORD init_width	Initial width.
WORD init_height	Initial height.

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.

Setting Window Information: wind_set( )

The **wind_set**( ) function sets certain specified fields for the window. To use it, see the following syntax summary.

WORD ret_code = **wind_set** (handle, flag, values[0]);

Input:

WORD handle	Window handle.	
WORD flag	What field to change:	
	2 WF_NAME	Address of a string containing the new window name. Uses values[0] and values[1].
	3 WF_INFO	Address of a string containing the new information line. Uses values[0] and values[1].
	5 WF_CXYWH	See **wind_get**( ).
	8 WF_HSLIDE	See **wind_get**( ).
	9 WF_VSLIDE	See **wind_get**( ).
	10 WF_TOP	See **wind_get**( ).
	14 WF_NEWDESK	Address of new default GEM Desktop.
		values[0] = low word of object tree.

	values[1] =	high word.
	values[2] =	starting object in tree to draw.
15 WF_HSLSIZE	See **wind_get**( ).	
16 WF_VSLSIZE	See **wind_get**( ).	

WORD values[4]

Output:

WORD ret_code If > 0, then no error. If = 0, then error exists.

Updating a Window: wind_update( )

The **wind_update**( ) function provides a resource locking mechanism for the AES and prevents other processes from doing a window update. While any process can still write to the screen, GEM AES attempts to provide a mechanism to allow a gentlemanly standard of conduct among cooperating processes. To use it, see the following syntax summary.

WORD ret_code = **wind_update** (code);

Input:

WORD code	0—END_UPDATE 1—BEG_UPDATE 2—END_MCTRL 3—BEG_MCTRL

Output:

WORD ret_code	If > 0, then no error. If = 0, then error exists.

THE OBJECT LIBRARY

So far we have examined two fundamental AES libraries: the Event Library and the Window Library. Now we need to discuss another critical set of functions in the GEM AES: the Object Library. While the importance of the other two libraries may seem obvious to you, the importance of the Object Library is more elusive. In fact, the Object Library provides some crucial connections between what appears on the screen and its representation within the GEM system. To make this clear, we will examine the usefulness of an object before presenting a definitive, byte-by-byte description of the contents of an object.

What Is a GEM Object?

Quite simply, a GEM *object* is a data structure contained in a GEM resource that describes what appears on the screen. Objects can be used to define plain boxes, boxes with text in them, icons, character text, and bit images. As you might suspect, GEM itself uses objects extensively. For example, a menu uses objects to represent what should appear on the screen, as does a dialog. While it is possible to write a GEM application without using any objects (which means that the application does not use resources), we don't recommend it, mainly because such an application will not be consistent with many of the features of the AES—that is, menus, dialogs, and so on will not look exactly the same. We believe that the overhead associated with using objects is worth it. An analogy may make the value of an object clearer.

Imagine that you are a decorator and want to add some bookshelves to a wall in order to store books and plants, and to hold knickknacks, and so on. You can, of course, get some cinder blocks and simple pine planking and build a functional student's bookcase, but this type of design won't provide the flexibility that you desire. Your goal is to be able to place this particular fixture to anyplace on the wall and to be able to link other units to it when you wish. In addition, you want to be able to change the internal shelving of each unit, so that you can have shelves that are wide apart or close together.

The GEM object provides the same level of flexibility and function as our imaginary wallhanging system. An object can be used to represent a number of graphic things on the screen (simple boxes, text, icons, and so on), which is similar to our basic wallhanging unit in the way it can contain a book, vase, plate, or any knickknack. Furthermore, an object may be placed anywhere on the screen, which is also like placing our wallhanging unit anywhere on a wall. Finally we can connect objects in what is called an *object tree* much as we can connect each unit of our wallhanger to form a new visual pattern on the wall. The GEM AES function, **objc_draw()**, displays an entire object tree.

Object Trees

GEM groups objects together to form trees, not just chains, by means of parent/child/sibling relationships between the objects in the tree. The importance of this is that the objects can contain other objects, defining complex *visual hierarchies* like boxes within boxes, adjoining areas of the screen contained within a larger area of the screen, menus within menus, and so on. For instance, all the levels of an application's menu bar can be represented by one object tree. Within that tree will be a subtree that defines the first level of the menu bar—that is, the material that appears on the screen when the application is first loaded. Another subtree defines the items within any of the first-level menus, and so forth. For an example, see the discussion DEMOMENU and Figure 5.6 in Chapter 5.

There are three pointers within the object data structure itself that identify the *next* sibling object in the tree, the *head* or first child of the current object, and the *tail* or last child of the current object. By using these three pointers, we can string together a number of objects into a tree. The tree is composed of a series of objects, each identifying where any children might be (which also tells us if any children exist). Figure 2.6 shows an example of an object tree that only shows the three pointers (we will fully define all the parts of an object shortly). Figure 2.6 is

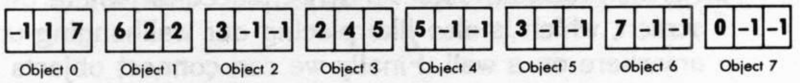


Figure 2.6: *Vector Representation of an Object Tree*

in the form of a long vector, as this is the way that an object tree exists in memory. See Figure 2.7 for a tree-like characterization of the same object tree showing the parent/child/sibling relationship.

WHAT IS A POINTER TO AN OBJECT?

By pointer, we really mean indices to the next object in the tree. In terms of the C language, an object tree is an array of structures of type object, where an integer index uniquely identifies any particular object in the array or tree. One of the consequences of defining tree pointers as indices is that all of the objects in a tree must be within the same 64K segment (on the Intel version of GEM).

Each object in Figure 2.6 shows a next sibling pointer, a pointer to the head of any children, and a pointer to the

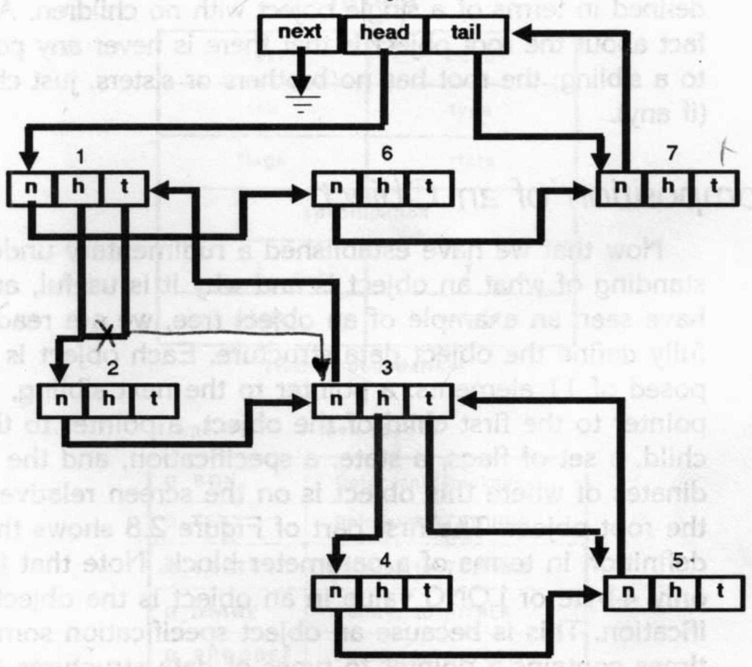


Figure 2.7: *Illustration of an Object Tree*

tail of children, respectively. Thus, the first object in Figure 2.6 (object 0) shows a – 1 or a null pointer to a sibling. Object 1 is the head of the children of object 0 (or, in other words, the first child of object 0), and object 7 is the last child of object 0. Similarly, object 1 shows that it has a sibling (object 6) and a child (object 2 is the first and only child of object 1). Now study Figure 2.7 to see how this example of an object tree looks when we take into account the relationships between the objects.

There is a main object, called the *root* (object 0), which has three children, two grandchildren, and two great-grandchildren. The root is always object 0, or the first object in the tree. There may be no other objects, but there is always a root. For instance, a single box on the screen can be defined in terms of a single object with no children. Another fact about the root object is that there is never any pointer to a sibling; the root has no brothers or sisters, just children (if any).

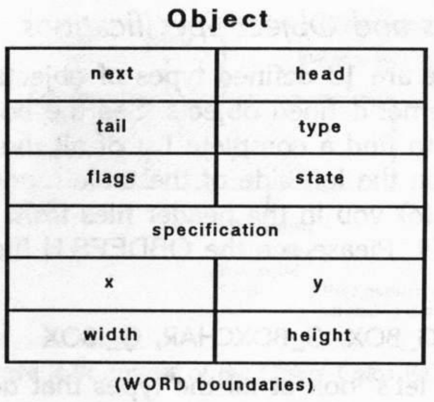
Composition of an Object

Now that we have established a rudimentary understanding of what an object is and why it is useful, and have seen an example of an object tree, we are ready to fully define the object data structure. Each object is composed of 11 elements: a pointer to the next sibling, a pointer to the first child of the object, a pointer to the last child, a set of flags, a state, a specification, and the coordinates of where this object is on the screen relative to the root object. The first part of Figure 2.8 shows this definition in terms of a parameter block. Note that the only 4-byte or LONG value in an object is the object specification. This is because an object specification sometimes contains a pointer to types of data structures that are not objects. This is an aspect of objects that we will discuss shortly.

We have already talked about the first three elements of an object, so let's examine the remaining eight.

Object Coordinates

First let's look at the last four WORD entries in the parameter block, shown in the top part of Figure 2.8. They contain the x, y, width, and height of where the particular object is to be placed on the screen. These values are absolute screen coordinates only for the root object of the tree. For the remaining objects, the coordinates are relative to the placement of the root object. This means that if we wanted to move the entire tree on the screen, all we would have to do is reposition the root object. GEM will redraw all of the children objects relative to the



if type is	then spec is
G_BOX	Color and thickness
G_TEXT	Pointer to TEDINFO
G_BOXTEXT	Pointer to TEDINFO
G_IMAGE	Pointer to BITBLK
G_PROGDEF	Pointer to APPLBLK
G_IBOX	Color and thickness
G_BUTTON	Pointer to string
G_BOXCHAR	Character, color & thick.
G_STRING	Pointer to string
G_FTEXT	Pointer to TEDINFO
G_FBOTEXT	Pointer to TEDINFO
G_ICON	Pointer to ICONBLK
G_TITLE	Pointer to string

Figure 2.8: *Object Structure*

new position of the root. This ease of repositioning an entire visual hierarchy is one of the reasons why objects are used so heavily in GEM.

Object Types and Object Specifications

There are 13 defined types of objects, including programmer-defined objects. See the bottom part of Figure 2.8 to find a complete list of all the object types. The names on the left side of the table (under "if *type* is") are defined for you in the header files from the GEM Developer's Kit. Please see the OBDEFS.H file in Appendix D.

Object Types: G_BOX, G_BOXCHAR, G_IBOX

First let's look at all the types that don't have pointers for object specifications. A G_BOX is an object type that produces a box on the screen at the specified coordinates (relative to the root, of course). A G_BOXCHAR is an object type that draws a single character within a box on the screen. A G_IBOX is an *imaginary box* that is actually invisible on the screen. This type of object is used to group the elements of an object, but you don't want to show the actual outline of a box around them. For instance, the Desktop file icon is a bit image plus some text (the file name) grouped together by an invisible box. The use of G_IBOX in this case eases the problem of object selection with **objc_find( )**.

All these box specifications have two associated WORD values: the color and thickness of the borders. The color is the low WORD of the LONG object specification field; it is described in Figure 2.9. The actual colors used in the object color WORD are defined in the top part of Figure 2.10. (You will notice that there are 4 bits assigned for each color in Figure 2.9. This allows color codes from 0 to 15, as seen in the top table of Figure 2.10.) All the names that start with an "D" indicate the darker shade of the color. For instance, DRED means "dark red."

The low byte of the high WORD of the object specification field associated with box object types defines the thickness of the borders of the box. The thickness of a

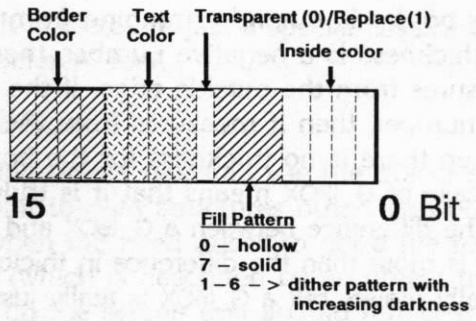

Border Color
Text Color
Transparent (0)/Replace(1)
Inside color
15
0 Bit
Fill Pattern
0 – hollow
7 – solid
1 – 6 – > dither pattern with increasing darkness

Figure 2.9: *Format of the Object Color WORD*

Colors

WHITE	0
BLACK	1
RED	2
GREEN	3
BLUE	4
CYAN	5
YELLOW	6
MAGENTA	7
WHITE	8
BLACK	9
DRED	10
DGREEN	11
DBLUE	12
DCYAN	13
DYELLOW	14
DMAGENTA	15

Flags

NONE	0X0000
SELECTABLE	0X0001
DEFAULT	0X0002
EXIT	0X0004
EDITABLE	0X0008
RBUTTON	0X0010
LASTOB	0X0020
TOUCHEXIT	0X0040
HIDETREE	0X0080
INDIRECT	0X0100

States

NORMAL	0X0000
SELECTED	0X0001
CROSSED	0X0002
CHECKED	0X0004
DISABLED	0X0008
OUTLINED	0X0010
SHADOWED	0X0020

Figure 2.10: *Colors, Flags, and States for Objects*

box's border is a number ranging from −127 to +128. If the thickness is a negative number, then the thickness measures from the outside edge. If the thickness is a positive number, then it measures from the inside edge. If it is 0, then there is no thickness for the box object, which in the case of G_IBOX means that it is truly an invisible box.

The difference between a G_IBOX and G_BOX with 0 thickness is more than the difference in thicknesses. A G_BOX is actually drawn, but a G_IBOX is really just a construct to hold together the object tree's visual hierarchy. A G_IBOX is a box by virtue of its name, whereas as G_BOX is really a box.

The high byte of the high WORD of the object specification field associated with box object types has meaning only if the object is a G_BOXCHAR type. In this case, the high byte contains the actual character to be displayed.

Object Types: G_STRING, G_BUTTON, G_TITLE

So far we have talked about three out of the 13 object types, limiting the discussion to those types in which the object specification is not a pointer. Now let's look at the simplest types that have a pointer for a specification.

G_STRING is an object type that defines where on the screen a simple text string should appear. Since there is no room within the data structure of the object itself to contain the string, however, GEM uses the object specification field to point to somewhere in memory where the string resides. Please note that in keeping with all strings in C, the string that the object specification of a G_STRING points to must be null-terminated.

The object type G_BUTTON also needs to point to a null-terminated string. This type is used to describe so-called buttons that have text within them, like the OK or Cancel buttons seen in many alert boxes.

Finally, the type G_TITLE describes the name that appears in menu titles. It is therefore just a special case of G_STRING, displayed in a different font.

The Rest of the Object Types

The remaining object types all use the object specification field to point to data structures other than objects. For instance, when the need arises to get a file name within a dialog (which is described by an object tree), the particular area on the screen that should reflect the file name information is controlled by a data structure called TEDINFO, which stands for "Text EDited INFOrmation." Another data structure that objects use is known as a BITBLK. BITBLKs are used to define graphics bit images. APPLBLKs and PARMBLKs allow the programmer to define her own data structures, which, of course, necessitates a lot of effort on the part of the programmer. Chapter 5 discusses programmer-defined objects in detail. Finally, there are ICONBLKs, which are used to define icons.

We will only be discussing in some detail the TEDINFO data structure, as we have not covered either bit images or icons in this book at all. Although we will show you the APPLBLK and PARMBLK data structure in a moment, we will wait until we discuss programmer-defined objects in Chapter 5 for any discussion of them.

TEDINFO *Structure*

The TEDINFO (Text EDited INFOrmation) structure provides a framework that allows the user to edit formatted text. There are four object types—G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOTEXT—that point to a TEDINFO structure that is shown in Figure 2.11. The structure contains a pointer to the text string, ptext (again there is no room in the structure itself to contain the actual string). Since TEDINFO structures allow edited input text, however, some rules must exist to determine how the text is to be displayed and what a valid input string should look like. Let's review these rules briefly.

ptext	
template	
valid	
font	reserved
just	color
reserved	thick
length	len_template

(WORD boundaries)

ptext – pointer to text string
template – pointer to text template
valid – pointer to validation string
length – number of characters in text string
len_template – number of characters in template

Figure 2.11: *TEDINFO Data Structure*

The text string pointed at by ptext may be blank, which is indicated either by a completely blank string or an at sign (@) in the first character position of the field. If the first character position is blank, the remaining characters may be anything—GEM AES will interpret the entire field as blank.

The template (template) that controls how the text appears on the screen points at another string that has underscored positions to indicate where the specified text is to be entered.

The final ingredient required is to describe the input validation field pointed at by valid. This string is a string of special characters that determines what input GEM will accept. Table 2.3 defines these validation characters.

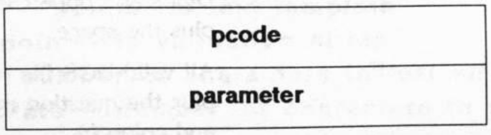
Validation Character	Input Allowed
9	Only the digits 0–9.
a	Upper- and lowercase alpha characters plus the space.
n	Digits, upper- and lowercase alpha characters, plus the space.
p	All the valid DOS path name characters, plus the backslash (\\) and the colon (:).
A	Only uppercase alpha characters, plus the space.
N	Digits and uppercase alpha characters, plus the space.
F	All valid DOS file name characters, plus the question mark (?), asterisk (*), and colon (:).
P	All valid DOS path name characters, plus the backslash (\\), colon (:), question mark (?), and the asterisk (*).
X	Anything.

Table 2.3: *TEDINFO Validation Characters*

For instance, what if the ptext text is "@" and the template is "Enter File Name: ________.___". In this case, only the template string would appear on the screen. Now to get the effects of allowing editing, we need to specify an input validation string, which is what valid points at. In this example, let's use "FFFFFFFFFFF" to represent the file name that the user is going to type in. If the user types "chapter1txt", then GEM shows the following on the screen: "Enter File Name: chapter1.txt".

The APPLBLK *and* PARMBLK *Structures*

To promote the greatest flexibility, GEM allows the programmer to define his own data structures for use within the object system. This is for the situation in which none of the defined object types really does what the programmer wants. The reason why we are deferring the discussion of this until Chapter 5 is that there are some tricky details involved with defining your own objects, and the coding examples illuminate those details nicely. At this time we will therefore simply illustrate the APPLBLK and PARMBLK structures in Figure 2.12 and 2.13, respectively.



pcode – pointer to the code to change object
parameter – optional LONG value as parameter

Figure 2.12: *APPLBLK Structure*

Building an Object Tree

In the last few pages, we have introduced you to the notion of a GEM object. As you will probably have gathered from our discussion, building them can be very tedious. For this reason, it is worth reemphasizing the value

ptree	
obj_index	old_state
state	obj_x
obj_y	obj_width
obj_height	clip_x
clip_y	clip_width
clip_height	parameter

ptree – pointer to object tree
clip_? – coordinates of current clipping rectangle
parameter – identical to APPBLK

Figure 2.13: *PARMBLK Structure*

of the Resource Construction Set (RCS). The RCS is an invaluable tool for dealing with all the resource data structures (objects, TEDINFOs, BITBLKs, and so on). You can build these structures by hand, but you only have to do that once to realize how difficult it can be.

Summary of Important Facts About Objects

We have presented only the bare essentials concerning the manipulation of objects and the other resource data structures. We do go into more detail about objects in the sample program in Chapter 5, but even so, there is a lot more to cover. Here, however, is a quick summary of some of the more important facts about objects that we have discussed.

— 1 Whenever an object tree is used as a parameter in any of the Object Library functions, it is a

LONG pointer to an array of objects.

— 2 The root of an object tree is always at index 0 of the array.

— 3 All pointers to objects consist of the pointer to an object tree plus the index into the object tree array.

— 4 Use the RCS.

— 5 The value for a NIL object index is −1, since 0 is a valid object index (the root).

Object Library Syntax Summary

This section is a short reference summary of all the functions in the Object Library.

Adding a Child Object: objc_add()

The **objc_add**() function adds an object as a child to the parent in an object tree. To use it, see the following syntax summary.

WORD ret_code = **objc_add** (obj_tree, parent, child);

Input:

LPTR obj_tree	Address of the object tree that is being added to.
WORD parent	Parent to which children are being added.
WORD child	Object to be added as a child to the specified parent.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Changing the Object State: objc_change( )

The **objc_change**( ) function changes a specified object's state within the confines of the specified clipping rectangle. To use it, see the following syntax summary.

```
WORD ret_code = objc_change (obj_tree, object, reserved,
                             clip_x, clip_y, clip_width,
                             clip_height, new_state,
                             redraw);
```

Input:

LPTR obj_tree	Address of the object tree that contains the object whose state will be changed.
WORD object	Object whose state will be changed.
WORD reserved	Must be 0.
WORD clip_x	X coordinate of the clipping rectangle.
WORD clip_y	Y coordinate.
WORD clip_width	Width of clipping rectangle.
WORD clip_height	Height of clipping rectangle.
WORD new_state	New state value (see Figure 2.10).
WORD redraw	0—Do not redraw object. 1—Redraw the changed object.

Output:

WORD ret_code	If $>$ 0, then no error. If $=$ 0, then an error occurred.

Deleting an Object: objc_delete( )

The **objc_delete**( ) function unlinks an object from the parent. To use it, see the following syntax summary.

WORD ret_code = **objc_delete** (obj_tree, object);

Input:

LPTR obj_tree	Address of the object tree that contains the specified object.
WORD object	Object to be deleted from tree.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Displaying an Object Tree: objc_draw( )

The **objc_draw( )** function draws the specified object, as well as any indicated descendants of the object (children, grandchildren, and so on). Note that siblings of the specified object are not included in the drawing list. The drawing is contained within a specified clipping rectangle. To use it, see the following syntax summary.

WORD ret_code = **objc_draw** (obj_tree, object, depth, clip_x, clip_y, clip_width, clip_height);

Input:

LPTR obj_tree	Address of the object tree that contains the specified object.
WORD object	Object that marks where in the object tree to begin drawing.
WORD depth	Relative to the above object, how many levels deep to draw: 0—Only the specified object. 1—Children of the specified object. 2—Children of the children of the specified object, and so on.
WORD clip_x	X coordinate of the clipping rectangle.

WORD clip_y	Y coordinate.
WORD clip_width	Width of the clipping rectangle.
WORD clip_height	Height of the clipping rectangle.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Changing the Editable Text Object: objc_edit( )

The **objc_edit**( ) function edits the text in the specified object, which must be either a G_TEXT or G_BOXTEXT (refer to the discussion of the TEDINFO data structure earlier in this chapter). To use **objc_edit**( ), see the following syntax summary.

WORD ret_code = **objc_edit** (obj_tree, object, character, char_idx, kind, &new_idx);

Input:

LPTR obj_tree	Address of object tree that contains the specified object.	
WORD object	Object containing the text to be edited.	
WORD character	Replacement character.	
WORD char_idx	Index of the next position in the text string.	
WORD kind		
	0 ED_START	Reserved.
	1 ED_INIT	Produce a formatted string by moving the input text into the template. Turn on the text cursor.

2 ED_CHAR	Make sure that the input characters are expected in the validation string of the TEDINFO structure. Update the text, and display the string.
3 ED_END	Turn off the text cursor.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD new_idx	Index into the text string that points to the next character position.

Locating an Object on the Screen: objc_find( )

The **objc_find( )** function finds out if there is an object under the mouse, given a starting location in the object tree. The function returns the index of the found object in the tree. To use **objc_find( )**, see the following syntax summary.

```
WORD obj_idx = objc_find (obj_tree, object, depth,
                          mouse_x, mouse_y);
```

Input:

LPTR obj_tree	Address of object tree that contains the specified object.
WORD object	Object in the tree to begin the search.
WORD depth	How deep into the tree:
	0—Only search the specified object.

	1—Search through the children of the specified object.
	2—Search through the grandchildren of the specified object.
WORD mouse_x	X coordinate of the mouse.
WORD mouse_y	Y coordinate.

Output:

WORD obj_idx	Index of the object in the object tree, if found. −1 if object not found.

Calculating an Object's Screen Coordinates: objc_offset( )

The **objc_offset**( ) function returns the actual screen coordinates of the specified object. The **objc_offset**( ) and **objc_find**( ) functions are inverse of each other. **objc_find**( ) takes an actual screen position and returns what object is displayed at that position, whereas **objc_offset**( ) takes an object and returns its actual screen position.

To use **objc_offset**( ), see the following syntax summary.

```
WORD ret_code = objc_offset (obj_tree, object, &obj_x,
                             &obj_y);
```

Input:

LPTR obj_tree	Address of object tree that contains the specified object.
WORD object	Compute this object's screen location.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD obj_x	X coordinate of the specified object relative to the screen.
WORD obj_y	Y coordinate.

Reordering the Children: objc_order( )

The **objc_order**( ) function reorders a child within the list of children of an object. To use it, see the following syntax summary.

WORD ret_code = **objc_order** (obj_tree, object, newpos);

Input:

LPTR obj_tree	Address of an object tree that contains the specified object.
WORD object	Object to be moved.
WORD newpos	New position in the list of children of the specified object. 0—Move to bottom of list. 1—Move to one from the bottom. 2—Move to two from the bottom, and so on. – 1—Move to top of the list.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

THE RESOURCE LIBRARY

Now that you have been introduced to the Object Library, let's look at the closely related AES Resource Library, which provides the services needed to manage a program's data or resources. In GEM, resources are the menus, dialogs, icons, bit images, and alerts of the application, and since these are represented by GEM objects, there is a strong relationship between an application's resources and its objects. In effect, the Resource Library provides more routines to manipulate GEM objects. In this section, we show you how to use the Resource Library routines to load and access the resources of your application.

This AES Library manipulates the resources contained in the .RSC file, which is usually created by the RCS. (Do not, however, confuse this file with the initials of the Resource Construction Set (RCS). They are related but distinct entities.) Only one .RSC file at a time is allowed for each application; if more than one is needed, the program can free the first and load the second. In this file are all the menus, dialogs, alerts, icons, and so on used by the application. Of course, the application may choose to create its own objects within itself, but this would be very difficult, and it would also limit the portability of the application. We vigorously urge the use of the RCS.

Thus, GEM separates the program into two distinct groups: the program itself and the data used by the program. The full usefulness of this separation is best seen when you want to modify the object trees that are the resources of the program. If the resources of the program were part of the program itself, then you would have to edit the program, recompile it, and relink it. But the fact that the resources are in a separate file (the .RSC file) means that when you need to change the resources, all you need to do is use the RCS to edit the .RSC file. No changes need to be made to the program itself. You can, of course, choose to merge your program's data with the program itself, but we do not recommend this since it turns the process of making minor changes to simple text into a major job.

An attractive side benefit to this separation is international portablity. Think of customizing your application for another country such as Japan. DRI has produced a version of the DRI GEM applications for the Japanese market with relatively little effort, because of the compartmentalization of the resources used by GEM applications. Most of the effort went into editing the text of the menus, dialogs, and alerts into Kanji (the host system already supported Kanji) using the RCS. None of the actual code had to be changed, just the text that the user sees.

THE 2 FILES THAT MAKE A GEM APPLICATION

Because GEM separates the program from its data in a separate resource file, most GEM applications reside in two files; the resource file (the .RSC file), and the application file itself (the .APP file). If the application uses the resources in the .RSC file then the .RSC file must be in the same directory as the .APP file.

The most used Resource Library functions are **rsrc_load()** and **rsrc_gaddr()**. The resource load function, **rsrc_load()**, finds the .RSC file on disk and loads all the object trees in memory. The **rsrc_gaddr()** function returns the address of a specified tree that was loaded by **rsrc_load()**. Once all the object trees for the application are loaded and available in memory, the application can address each tree as needed and use the information stored within each tree. Notice that **rsrc_load()** returns only an error code and nothing else, implying that only one resource file can get loaded for each application at a time. If you need to switch resources, you must use the **rsrc_free()** function to free the used memory in the already loaded .RSC file before loading the next .RSC file.

There is a linkage between the .RSC file and the application via another file created by the RCS. The RCS will produce, upon direction, an C include file (a file with a file type of .h) for use in the application. The include file names the indices of the trees and objects in the resource file. (For example, see the listing of DEMO.H in Chapter 5.) You get to name the objects (see Appendix C) when you build the object using the RCS. These names are just C defined names of the object indices. If you do not name the object, then the include file will not contain the object. After naming the objects, you can use names instead of numbers in the AES functions that take object indices. This should minimize the kinds of errors that result from using the wrong object index.

Let's look at the way DEMO uses the Resource Library routines. After the .RSC file is loaded, DEMO gets the address of the object tree that describes its menu bar, and calls **menu_bar**( ) to display the menu bar. Similarly, when one of DEMO's dialogs needs to be displayed (in order to allow the user to save the screen, open a file, or change the pencil style, for example), DEMO calls **rsrc_gaddr**( ) with the index of the object that describes the dialog in order to get the address in RAM where the tree is stored. This pointer is then used by the **objc_draw**( ) to display the dialog and by **form_do**( ) to process the user input while the dialog is active.

One of the more common mistakes in using the Resource Library and the Object Library is to confuse the name of a tree with the name of an object in the tree. Essentially, this means that you are using the wrong object index (instead of confusing the tree name with the object name, you can also, of course, simply use the wrong object name). In the case of using the wrong object index, you could end up overindexing the object structure and getting something somewhere in RAM. This probably won't do you any good; in fact, you could end up rebooting the system. There is no real way to predict the behavior when you over-index. You could get lucky and index into another object tree, but then you would be using the wrong tree.

The material in Chapter 5 shows how most of the major routines in the Resource Library are used.

Resource Library Syntax Summary

This section is a short reference summary of all the routines in the Resource Library.

Freeing Memory Allocated for Resources: rsrc_free( )

The **rsrc_free**( ) function releases the memory allocated by **rsrc_load**( ) and is used when the program wants to load another set of resources. If another set of resources is loaded, the old set that is already loaded will be discarded. To use **rsrc_free**( ) see the following syntax summary.

WORD ret_code = **rsrc_free** ;

Input:

None

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Getting the Address of the Loaded Resource: rsrc_gaddr( )

The **rsrc_gaddr**( ) function gets the address of the specified data in the structure after the resource is loaded in memory. To use it, see the following syntax summary.

WORD ret_code = **rsrc_gaddr** (type, index, &data);

Input:

WORD type	The type of object or data pointed at by index, as follows: 0—Tree. 1—OBJECT. 2—TEDINFO. 3—ICONBLK.

	4—BITBLK.
	5—String.
	6—Image data.
	7—Object specification.
	8—Pointer to text (TEDINFO).
	9—Pointer to text template (TEDINFO).
	10—Pointer to text validation string (TEDINFO).
	11—Pointer to mask of icon bit image (ICONBLK).
	12—Pointer to data of icon data image (ICONBLK).
	13—Pointer to text of icon (ICONBLK).
	14—Pointer to bit image (BITBLK).
	15—Address of a pointer to a free string.
	16—Address of a pointer to free image.
WORD index	Index of the data.
Output:	
WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
LPTR data	Address of data in memory.

Loading a Resource File into Memory: rsrc_load()

The **rsrc_load**() function loads the specified .RSC file into memory. To use this function, see the following syntax summary.

```
WORD ret_code = rsrc_load (RSCfile);
```

Input:

LPTR RSCfile	Address of .RSC file name string.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Changing Object Tree Coordinates: rsrc_obfix( )

The **rsrc_obfix( )** function changes the form of the coordinates in the specified object tree from character coordinates to pixel coordinates. Character coordinates are created by the RCS because it doesn't know the device resolutions when the resource file is being created. **rsrc_load( )** uses the **rsrc_obfix( )** function to convert the coordinates.

To use this function, see the following summary.

```
WORD dummy = rsrc_obfix (tree, object);
```

Input:

LPTR tree	Address of tree that contains the object.
WORD object	Index of object.

Output:

dummy	Always 1.

Storing Free Strings: rsrc_saddr( )

The **rsrc_saddr( )** function stores the values of either the free string or free image addresses in the specified object to another place in the object tree. To use this function, see the following summary.

```
WORD ret_code = rsrc_saddr (type, index, saddr);
```

Input:

WORD type	Type of data:

		15—Address of a pointer to a free string.
		16—Address of a pointer to a free image.
	WORD index	Index of destination in the data.
Output:		
	WORD ret_code	If > 0, then no error. If = 0,then an error occurred.
	LPTR saddr	Address of the data.

THE MENU LIBRARY

We are now ready to discuss one of the more common uses of an object tree, the GEM menu. Each application that uses the GEM standard interface can make use of the typical GEM menu bar on the top of the physical screen. GEM uses menus to provide the options available to each application. For instance, in GEM Draw the menu bar displays options that allow the user to change the text font and size, change the fill pattern, arrange the objects on the Draw screen, write the window to a file, use the desk accessories, and more.

By choosing to use the GEM Menu Library, an application gains by having a consistent interface to follow. The Menu Library routines make it possible to

— Display the appropriate menu bar for the application.

— Display an item that cannot be selected by displaying it at a dimmed brightness level. Items that can be selected are shown at normal brightness.

— Display menu text.

GEM menus are known as *drop-down menus* because when the user moves the mouse over the menu bar, the Screen Manager drops the entire menu down onto the screen. In contrast, the Macintosh uses *pull-down menus,* which work by having the user click on the desired menu, and, holding the button down, move through the menu highlighting each pointed-at item. By releasing the button the user selects the last highlighted item. Thus, on the Macintosh, the menu is displayed as long as the button remains depressed, whereas GEM menus are visible until the user moves the mouse out of the menu, either into another menu or to another part of a screen. GEM menus are also different in that the mouse button is used to to select a menu item.

The Screen Manager handles all of the user interaction with the menu. This includes the saving of the area of the screen that is overwritten when the menu is dropped down, and restoring it again when the user is finished with the menu. One of the rules of writing GEM applications concerns the size of the menu itself. Because the Screen Manager uses a fixed buffer the size of 1/4 of the screen, no menu can exceed that size. This means that when you build the menus for your application using the Resource Construction Set, you will have to be careful not to build a menu that is too large. If you find that you need a larger menu, then cut it up into smaller submenus.

Using Menu Functions

Of the six menu functions, the three most interesting ones are **menu_bar( )**, **menu icheck( )**, and **menu_ienable( )**. All programs that use menus will have to use the **menu_bar( )** function as this is the only function that displays the menu bar on the screen. One of the first things that DEMO does in the **demo_init( )** routine is to load the resource file and later display the object tree that describes the menu bar (see Listing 5.3 and Listing 5.5). Unfortunately, DEMO does not use the other two menu routines, so we we will have to discuss these without the benefit of having an example to refer to.

One of the conventions of the GEM menu system is that menu items that can be selected appear in normal brightness on the screen. If the menu item is not selectable, then by convention the menu item should appear at a dimmed brightness level. GEM does not enforce this because it cannot know when a menu item is selectable or not for your GEM application; only the application itself can know that. But GEM does provide a function called **menu_ienable( )** to aid the application in dimming or brightening a menu item.

As the application receives information from the Screen Manager concerning the user's choice of menu items, the application knows whether or not each menu item in each menu is selectable. When an item is no longer selectable, the application uses **menu_ienable( )** to signal GEM to dim that item's brightness the next time that it appears on the screen. The reverse occurs when the menu item becomes selectable again. The initial brightness of a menu item is governed by the value of its object flag and state (see Figure 2.10). Thus, **menu_ienable( )** changes the state and flag of the object that describes the menu item. The Screen Manager uses that information when it displays the menu the next time that the user chooses the menu.

It's interesting to note here that while Chapter 5's DEMO program does not use **menu_ienable( )** to highlight the menu list, DEMO does in fact highlight the menu list without using the Menu Library. Quite simply, all that DEMO does is to modify directly the object flag and state fields of the object that describes the menu item. DEMO thus avoids the overhead of using an additional AES function. See Chapter 5 for a more complete discussion.

The other major menu function is **menu_icheck( )**, which either places a check mark next to a selected menu item or erases a check mark. The use of check marks in menus is a way that GEM allows the application to show to the user what options are currently active. For instance, in GEM Draw, if the user wants boldfaced type, then Draw places a check mark next to that menu item in the Text menu. Every time that the user selects the Text menu, he will see the check mark next to the active items that he has selected. If the user changes the selection, then Draw erases the old check mark and displays a check mark next to the newly selected item.

Menu Library Syntax Summary

This section is a short reference guide to all the functions of the Menu Library. All the object trees that are referenced in the summaries must be loaded using the **rsrc_load**( ) function (documented in the section on the Resource Library).

Displaying the Menu Bar: menu_bar( )

The **menu_bar**( ) function shows or erases the menu bar. To use this function, see the following syntax summary.

WORD ret_code = **menu_bar** (obj_tree, action);

Input:

LPTR obj_tree	Address of object tree that produces this menu.
WORD action	0—Erase the menu bar. 1—Show the menu bar.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Displaying or Erasing a Check Mark on a Menu Item: menu_icheck( )

The **menu_icheck**( ) function places or erases a check mark next to the specified menu item in order to indicate menu selection. To use this function, see the following syntax summary.

```
WORD ret_code = menu_icheck (obj_tree, item, check);
```

Input:

LPTR obj_tree	Address of the object tree that produces this menu.
WORD item	Menu item identifier.
WORD check	0—Erase a visible check mark next to specified item, or do not display a check mark next to item. 1—Place a check mark next to item.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Enabling or Disabling a Menu Item: menu_ienable( )

The **menu_ienable**( ) function enables or disables a menu item. In this way, the application controls what menu items can be selected. To use this function, see the following syntax summary.

```
WORD ret_code = menu_ienable (obj_tree, item, enable);
```

Input:

LPTR obj_tree	Address of object tree that produces this menu.

WORD item	Menu item identifier.
WORD enable	0—Disable the menu item by dimming its characters. 1—Enable the menu item by displaying it in normal brightness.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Registering a New Desk Accessory: menu_register( )

The **menu_register**( ) function adds a name to the desk accessory Desk menu list. Since only six desk accessories can be loaded, only six menu items may appear in the Desk menu. To use this function, see the following syntax summary.

```
WORD menu_id = menu_register (DA_id, DA_txt);
```

Input:

WORD DA_id	Desk accessory ap_id.
LPTR DA_txt	Address of desk accessory's Desk menu test string.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD menu_id	Desk accessory's menu item identifier (0 – 5): – 1 means no room in the Desk menu for this item.

Replacing the Text of a Menu Item: menu_text( )

The **menu_text**( ) function replaces the text of the specified menu item. To use this function, see the following syntax summary.

WORD ret_code = **menu_text** (obj_tree, item, text);

Input:

LPTR obj_tree	Address of object tree that produces this menu.
WORD item	Menu item identifier.
LPTR text	Address of a text string to replace the string currently in the menu object tree. The new string cannot be longer than the old string.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Displaying Menu in Reverse or Normal Video: menu_tnormal()

The **menu_tnormal**() function displays the menu title in reverse or normal video modes. To use this function, see the following syntax summary.

WORD ret_code = **menu_tnormal** (obj_tree, title, norm_vid);

Input:

LPTR obj_tree	Address of the object tree that produces this menu.
WORD title	Menu title identifier.
WORD norm_vid	0—Display the title in reverse video. 1—Display the title in normal video.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

THE FORM LIBRARY

So far, we have studied objects, resources, and menus, all of which are pieces of the jigsaw puzzle of AES resources. We need to complete the puzzle by looking at the Form Library, which essentially presents two types of frameworks to handle user interaction. The two types are dialogs and alerts. First, let's deal with the simplest one, alerts.

Alerts

An *alert* is usually issued when an error condition is encountered in the application. For instance, the application might get an error trying to load the .RSC file; perhaps the file couldn't be found. In the typical application environment, the program would probably print out an error message before terminating. An alert is a standardized form of a message that serves the same function. In DEMO, alerts are used in exactly this fashion, as you can see if you look at the use of the **form_alert**( ) at the top of Listing 5.3, where DEMO trys to load the .RSC file and fails. DEMO uses an alert to notify the user that it cannot successfully load the .RSC file.

The message that an alert displays has a particular format, as follows:

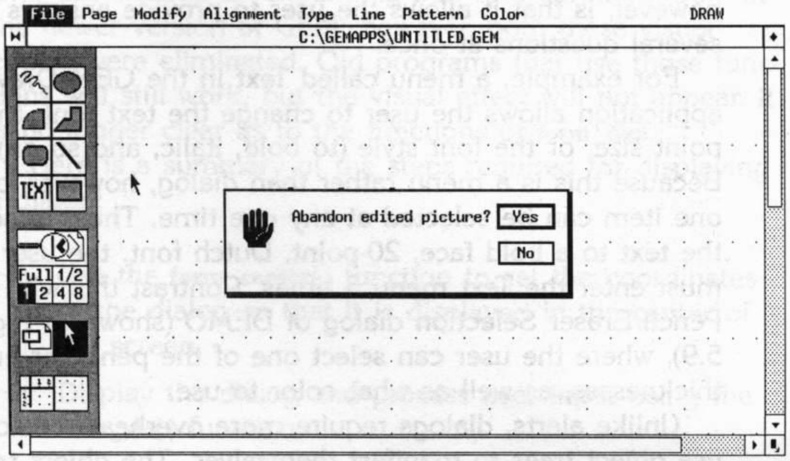
```
[icon #][message text][exit buttons]
```

Each part of the alert message is separated by square brackets. The icon # refers to the fact that there are four defined icons for an alert:

0—no icon
1—NOTE icon

2—WAIT icon
3—STOP icon.

A NOTE icon should be used when you just want to pass some information on to the user. An alert that uses a WAIT icon is forcing the user to pause in processing, while a STOP alert indicates to the user that bad things could happen if the user continues. STOP alerts are used to indicate that data could be lost if the user continues. See Figure 2.14 for an example of an alert that is used by Draw to warn a user that the current picture has not been saved.



Abandon edited picture?

Figure 2.14: *Sample Alert from Draw*

There is only a small space for the message in any alert: 5 lines of text with at most 40 characters per line. To force line breaks, use the vertical bar (the logical OR symbol).

An exit button is the G_BUTTON object type, and it returns the user to the application that invoked the alert. The text for each exit button is limited to less than 20 characters. There can be at most three exit buttons used, with the text for each button separated by a vertical bar.

Dialogs

A *dialog* is a superset of an alert. Where an alert usually tells the user something and allows the user to make some restricted, predefined response, a dialog is more flexible. Many dialogs allow the user to enter some text (see the section on the File Selector Library later in this chapter) and edit it. The chief characteristic of a dialog, however, is that it allows the user to provide answers to several questions at once.

For example, a menu called Text in the GEM Draw application allows the user to change the text font, the point size, or the font style (to bold, italic, and so on). Because this is a menu rather than dialog, however, only one item can be selected at any one time. Thus, to set the text to a bold face, 20-point, Dutch font, the user must enter the Text menu 3 times. Contrast this with the Pencil/Eraser Selection dialog of DEMO (shown in Figure 5.9), where the user can select one of the pencil or eraser thicknesses, as well as what color to use.

Unlike alerts, dialogs require more overhead. Dialogs use object trees to manifest themselves. The object tree format is covered in detail in Chapter 5 along with examples of dialogs used by DEMO. The point is that alerts can be as easy to use as the standard C output routine, **printf**( ), whereas dialogs require more planning and more steps.

Dialogs are typically displayed in the center of the screen by means of the **form_center**( ) function (see the syntax summary section). It is important to note that displaying dialogs in the center of the screen is really not just a convention. The root of a dialog object tree is set to 0, which means that if you do not use **form_center**( ), the dialog will appear in the left-hand corner of the screen. Since the whole point of a dialog is to grab the user's attention, the upper left corner of the screen is not generally where you will want the dialog to appear. The **form_center**( ) call sets the coordinates of the root of the dialog object tree to make sure that the dialog appears in the center of the screen. Note that instead of using **form_center**( ), you can directly set the coordinates of the root to any place on the screen at which you want the dialog to appear.

form_dial( ) is another function that has been modified by DRI because of the Apple disagreement. In the past, **form_dial** showed an expanding box or a shrinking box. In the newer version of GEM, however, both these visual effects were eliminated. Old programs that use those functions will still work, but the visual effect will not appear. It is no longer clear as to the functions of **form_dial**.

Here is a summary of the steps required for displaying a dialog:

— Use the **form_center**( ) function to set the coordinates of the dialog so that it is displayed in the center of the screen.

— Display the dialog and process user input using the **form_do**( ) function.

Form Library Syntax Summary

This section is a short reference summary of all the functions in the Form Library.

Displaying an Alert: form_alert( )

The **form_alert**( ) function displays the specified string as a GEM alert. The string must conform to the format described above. To use **form_alert**( ), see the following syntax summary.

WORD exit_button = **form_alert** (def_button, string);

Input

WORD def_button	The default exit button: 0—No default button 1—First exit button 2—Second exit button 3—Third exit button.
LPTR string	Address of the alert string.

Output:

WORD exit_button	Identifies which exit button was selected.

Centering the Dialog on the Screen: form_center( )

The **form_center**( ) function returns the appropriate coordinates such that when the specified object is drawn, it is centered on the screen. To use it, see the following syntax summary.

WORD dummy = **form_center** (tree, &X, &Y, &width, &height);

Input:

LPTR tree	Address of object that describes object tree to be centered.

Output:

WORD dummy	Always 1.
WORD X	X value of centered coordinates.
WORD Y	Y value.
WORD width	Width of centered object.
WORD height	Height.

Reserving and Freeing Screen Space for Dialogs: form_dial

The **form_dial()** performs one of two tasks as defined by one of its input arguments, flag. To use **form_dial()**, see the following syntax summary.

WORD retcode = **form_dial** (flag, smallx, smally, smallw, smallh, bigx, bigy, bigw, bigh);

Input:

WORD flag	The action of **form_dial()**.
	0—Reserve the screen space for dialog
	3—Free reserved screen space.
WORD smallx	X coordinate of smallest box.
WORD smally	Y coordinate.
WORD smallw	Width.
WORD smallh	Height.
WORD bigx	X coordinate of largest box.
WORD bigy	Y coordinate.
WORD bigw	Width.
WORD bigh	Height.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Executing the Displayed Dialog: form_do()

The **form_do()** function processes all the user input for the displayed dialog, and only returns to the calling program when the user selects one of the dialog's exit buttons. To use **form_do()**, see the following syntax summary.

WORD exit_obj = **form_do** (tree, object);

Input:

LPTR tree	Address of tree that contains the displayed dialog.
WORD object	The starting object within this tree that accepts user input (must be an editable text field). If 0, then displayed object doesn't contain any editable text fields.

Output:

WORD exit_obj	Object index of the object that caused the exit from the dialog.

Displaying DOS Errors: form_error( )

The **form_error**( ) function displays an alert identifying a DOS error. To use it, see the following syntax summary.

WORD exit_button = **form_error** (DOS_code);

Input:

WORD DOS_code	DOS error code.

Output:

WORD exit_button	Exit button identifier: 1—First exit button. 2—Second exit button. 3—Third exit button.

THE APPLICATION LIBRARY

At this point we have examined all of the major AES libraries. The remaining libraries provide some key functions for a GEM application, although not all of the functions in each of these minor libraries are immediately useful. Let's start with the Application Library.

The seven Application Library routines provide for

— Sharing of the AES libraries with other processes (**appl_init( )** and **appl_exit( )**).

— Using the message system to send or receive messages to or from other processes (**appl_read( )**, **appl_write( )**, and **appl_find( )**).

— Recording or playing back a GEM AES session (**appl_trecord( )** and **appl_tplay( )**).

By far the most important function in the Application Library is **appl_init( )**. Each application that uses the AES must register itself with the AES by calling **appl_init( )**. The **appl_init( )** function reserves some space in the Screen Manager's data area that the Screen Manager then uses for application-specific information. The reason for this is the limited multitasking environment of GEM. All multitasking operating systems allocate a process descriptor in memory for the very same reasons.

The second most useful functions of the Application Library are **appl_read( )** and **appl_write( )**. These functions are used to send or receive messages between processes.

Application Library Syntax Summary

This section is a short reference section to all the functions in the Application Library.

Cleaning Up an Application Environment: appl_exit( )

The **appl_exit( )** function cleans up the resources reserved when the application first registered with the AES by calling **appl_init( )**. To use it, see the following syntax summary.

WORD ret_code = **appl_exit** ( );

Input:

None

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error ocurred.

Finding the ID of Another Active Application: appl_find( )

The **appl_find**( ) function gets the ap_id of another active GEM application. This function is used in the procedure to establish communication with another application, because before an application can send messages to another application, both applications need to know about each other (each application needs to have the other's ap_id). To use **appl_find**, see the following syntax.

WORD search_id = **appl_find** (ap_name);

Input:

LPTR ap_name	Address of a null-terminated file name of an application. The file name must be eight characters long (unused characters must be filled by blanks).

Output:

WORD search_id	The ap_id associated with the named application. If the application is not running, then −1 is returned.

Registering an Application: appl_init( )

The **appl_init**( ) function registers the application with the AES, which initializes any resources that are specific to this invocation of the application. To use it, see the following syntax.

```
WORD ap_id = appl_init ( );
```

Output:

WORD ap_id	The ap_id associated with this invocation of the application.
	If the application cannot be registered with AES, then ap_id = −1 on return.

Reading a Message: appl_read( )

The **appl_read( )** function reads a message of specified length into the specified buffer. This function is usually used to read messages that are larger than the GEM standard of 16 bytes. Remember that the first 16 bytes of the message are sent as a message event (see the discussion of **evnt_mesag( )** in the Event Library section). To use **appl_read( )**, see the following syntax summary.

```
WORD ret_code = appl_read (pip_ap_id, length,
                    read_buffer);
```

Input:

WORD pipe_ap_id	ap_id of the application process that owns the pipe to read.
WORD length	Number of bytes to read from the pipe.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
read_buffer	Address of a buffer to contain the message.

Replaying a Recorded AES Session: appl_tplay( )

The **appl_tplay( )** function replays a previously recorded set of user GEM AES activity. To use it, see the following syntax summary.

```
WORD dummy = appl_tplay (LPTR ap_tpmem, WORD
                         ap_tpcount, WORD ap_tpscale);
```

Input:

LPTR record	Address of buffer that holds the recording of the user's actions.
WORD num_replay	Number of user's actions to replay.
WORD speed	Speed of replay (1–10,000).

Output:

WORD dummy	Always 1.

Recording an AES Session: appl_trecord( )

The **appl_record**( ) function produces a record of the user activity with the application. The actual number of events that are recorded is dependent on the size of the buffer and the number of events that the user creates. To use it, see the following syntax.

```
WORD num_evnts = appl_trecord (LPTR ap_trmem, WORD
                               ap_trcount);
```

Input:

LPTR record_buf	Address of a buffer to hold the recording of the user's actions. Each recorded action requires 6 bytes, a WORD and a LONG.
	WORD type => 0—timer event
	1—button

	2—mouse	
	3—keyboard	
	The LONG value depends on the type of event:	
	timer—	Number of elapsed milliseconds.
	button—	The low word is the button state (0/1, up/down). The high word is the number of clicks.
	mouse—	The low word is the mouse's X coordinate, and the high word is the mouse's Y corrdinate.
	keyboard—	The low word is the character that was typed, and the high word is the keyboard state.
WORD capacity	The number of events that can be stored in the buffer (size of buffer or 6 bytes).	

Output:

WORD num_evnts	The actual number of events that was recorded.

Sending a Message: appl_write( )

The **appl_write**( ) function sends a message of specified length to a specified application process. To use it, see the following syntax summary.

```
WORD ret_code = appl_write (write_id, length,
                            write_buffer);
```

Input:

WORD write_id	The ap_id of the destination application process.
WORD length	Number of bytes in the message.
LPTR write_buffer	Address of a buffer that contains the message.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

THE GRAPHICS LIBRARY

GEM AES provides a set of eight routines to assist you in manipulating rectangles on the screen. Examples of these routines include **graf_rubberbox( )**, which produces an expanding and contracting box, **graf_dragbox( )**, which drags a box around the screen, or **graf_mbox( )**, which moves a box on the screen. Two routines that used to be in this library—namely, **graf_growbox( )** and **graf_shrinkbox( )**—have been removed because of the Apple-DRI imbroglio. You will notice that DEMO still uses these routines, but they have no action; no expanding box or shrinking box appears on the screen when DEMO is used on the latest version of GEM.

One important contribution made by the Graphics Library is to return the VDI handle to the currently open screen workstation via the **graf_handle( )** function. The main use for **graf_handle( )** is to uniquely identify the open physical workstation, although **graf_handle** also returns the size of the system font on the open workstation. This handle is converted to a virtual workstation identifier by the VDI open virtual workstation function, **v_opnvwk()**. Chapters 4 and 5 provide examples of the **graf_handle()** function.

Another useful Graphics Library function is the **graf_mouse()** routine that is used to change the mouse cursor form or mouse form. For instance, when the user selects the text mode in GEM Draw, the representation of the cursor changes from an arrow to cross hairs. Thus, the mouse form often signals a particular function. The Graphics Library also provides a function that returns the current mouse position, the current button, and the current keyboard state (**graf_mkstate()**).

The last group of Graphics Library functions that we will discuss deals with movement of a box on the screen. We have already mentioned the functions in this group. **graf_dragbox()** is used to get the effect of dragging a rectangle around the screen. In the GEM Desktop, for instance, when a user drags a folder to another window by clicking on the folder icon, holding down the mouse button, and moving the mouse to the next window, the folder looks like it is dragged across the screen. **graf_dragbox()** produces this effect.

Another useful effect is that of the rubber box outline. GEM Draw uses this most effectively when the user wants to draw a rectangle on the screen. The user holds the button down and moves the mouse on the screen. Draw shows a box that has one vertex at the point where the mouse button was first clicked and the opposite vertex at the current position of the mouse. As long as the button is depressed, Draw continues to show a box that expands and contracts as the mouse moves across the screen; hence the term "rubber box."

Graphics Library Syntax Summary

This section is a short reference summary of all the functions in the Graphics Library.

Dragging a Box Around the Screen: graf_dragbox()

The **graf_dragbox()** function drags a specified box within a specified boundary rectangle. The mouse cursor position is kept constant relative to box's upper left corner throughout the dragging. To use this function, see the following syntax summary.

```
WORD ret_code = graf_dragbox (width, height, startx,
                              starty,boundx, boundy,
                              boundw, boundh, &finishx,
                              &finishy);
```

Input:

WORD width	Width of box being dragged.
WORD height	Height of box.
WORD startx	X coordinate of starting point.
WORD starty	Y coordinate.
WORD boundx	X coordinate of boundary rectangle.
WORD boundy	Y coordinate.
WORD boundw	Width of boundary rectangle.
WORD boundh	Height of boundary rectangle.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD finishx	X coordinate of the box at the end of the dragging.
WORD finishy	Y coordinate.

Getting the Opened Workstation Handle: graf_handle()

The **graf_handle()** function returns the VDI handle for the currently opened physical workstation, as well as the system font character size. To use this function, see the following syntax summary.

```
WORD vdi_handle = graf_handle (&char_width, &char_height,
                                     &box_width, &box_height);
```

Input:

WORD char_width	Width of a character cell used in the system font (in pixels).
WORD char_height	Height of a character cell used in the system font (in pixels).
WORD box_width	Width of a square box that holds a system font character (in pixels).
WORD box_height	Height of a square box that holds a system font character (in pixels).

Output:

WORD vdi_handle	GEM VDI handle.

Moving a Box: graf_mbox( )

The **graf_mbox**( ) moves a box from the source coordinates to the destination without changing the size of the box. To use this function, see the following syntax summary.

```
WORD ret_code = graf_mbox (width, height, from_x,
                    from_y, to_x, to_y);
```

Input:

WORD width	Width of a box at (from_x, from_y).
WORD height	Height of box.
WORD from_x	X coordinate of box.
WORD from_y	Y coordinate.
WORD to_x	Destination x coordinate.
WORD to_y	Destination y coordinate.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Finding the Current Mouse Position: graf_mkstate( )

The **graf_mkstate**( ) function gets the current mouse position, button state, and keyboard state. To use this function, see the following syntax.

```
WORD dummy = graf_mkstate (&mouse_x, &mouse_y,
                           &mouse_state, &kb_state);
```

Output:

WORD dummy	Always 1.
WORD mouse_x	X coordinate of current mouse position.
WORD mouse_y	Mouse y coordinate.
WORD mouse_state	The current mouse button state. 0x0001—Button on left. 0x0002—Second button from left. 0x0003—Third button from left. etc. 0—Means button up. 1—Means button down.
WORD kb_state	State of the keyboard's Shift keys, Ctrl key, and Alt key. (0—key up, 1—key down) 0x0001—Right Shift 0x0002—Left Shift 0x0004—Ctrl 0x0008—Alt

Changing the Mouse Form: graf_mouse( )

The **graf_mouse**( ) function changes the mouse form into the specified form, hides the mouse form, or shows it. To use this function, see the following syntax summary.

WORD ret_code = **graf_mouse** (form, mfdb);

Input:

WORD form	Mouse form code: 0—Arrow. 1—Text cursor (vertical bar). 2—Hour glass. 3—Hand with pointing finger. 4—Flat hand, extended fingers. 5—Thin cross hairs. 6—Thick cross hairs. 7—Outline cross hairs. 255—Use form described in MFDB. 256—Hide mouse form. 257—Show mouse form.
LPTR mfdb	Address of a 35-word mouse form definition block, which specifies a user-defined mouse form.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.

Drawing a Rubber Box: graf_rubberbox( )

The **graf_rubberbox**( ) function draws a rubber box on the screen. To achieve this, the user presses the mouse button, and drags the rubber box outline. The upper left corner is fixed at the point on the screen where the user first depressed the button, but the lower right corner can be dragged. When the user lets up on the button, the new box size is returned.

```
WORD ret_code = graf_rubberbox (box_x, box_y, min_width,
                                min_height, &last_width,
                                &last_height);
```

Input:

WORD box_x	X coordinate of box.
WORD box_y	Y coordinate.
WORD min_width	Minimum width (in pixels).
WORD min_height	Minimum height (in pixels).

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD last_width	Width of box when user lets button go.
WORD last_height	Height of box when user lets button go.

Finding the Center of the Slider: graf_slidebox( )

The **graf_slidebox( )** returns the center of the slider as it moves within the parent box. The user holds down the button while moving the mouse in the slider area. This moves the slider around the slider box.

```
WORD cent_slide = graf_slidebox (obj_tree, parent, object,
                                 direction);
```

Input:

LPTR obj_tree	Address of object tree containing the slider and the parent.

WORD parent	Index of the parent in the object tree.
WORD object	Index of the slider object.
WORD direction	0—Horizontal slider movement.
	1—Vertical slider movement.

Output:

WORD cent_slide	Location of the center of the slider relative to the parent. If slider moves horizontally within the parent box, then 0 is the leftmost position and 1000 is the rightmost. If the slider moves vertically within the parent box, then 0 is the topmost position, while 1000 is the bottom.

Tracking Mouse Movement in a Marked Box: graf_watchbox( )

The **graf_watchbox( )** function allows a box to be marked and then tracks the mouse movement into and out of this box. The user must have depressed the button and held it down throughout this call. **graf_watchbox( )** returns the final position relative to the watchbox of the mouse when the button is released.

```
WORD mouse = graf_watchbox (obj_tree, object,
                            inner_state, outer_state);
```

Input

LPTR obj_tree	Address of the object tree that produces the specified box.
WORD object	Index of object in object tree.
WORD inner_state	State of box when mouse is inside:

	0x0000—NORMAL
	0x0001—SELECTED
	0x0002—CROSSED
	0x0003—CHECKED
	0x0004—DISABLED
	0x0005—OUTLINED
	0x0006—SHADOWED.
WORD outer_state	State of box when mouse is outside.
Output:	
WORD mouse	Mouse cursor position when user lets go of the button. If 0, then the mouse is outside the box. If 1, then the mouse is inside the box.

THE FILE SELECTOR LIBRARY

Because GEM is an operating environment and not an operating system, it does not supply all the services that an application might need. In particular, GEM does not provide a file system but instead relies on the underlying DOS file system. There are therefore no direct file system functions in GEM, such as functions for opening, reading, or writing disk files. There is, however, one function in the File Selector Library—a function called **fsel_input()**—whose purpose is to display the File Selector dialog box (see Figure 2.15). **fsel_input()** provides an adequate interface between the user and the underlying file system since most GEM users need only to identify file names, leaving GEM to do the reading or writing.

When an application requires the user to provide a file name, the application calls **fsel_input()** to display the contents of a directory where the user can select one of the

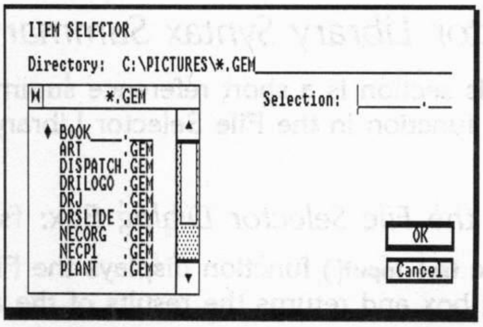


Figure 2.15: *File Selector Dialog*

existing files or create a new file. For example, GEM Draw allows the user to save the work area into a file by clicking on the File menu's "Save As" item. GEM Draw then invokes **fsel_input**( ) with the pictures directory as input.

The File Selector dialog box allows the user to choose any displayed file either by clicking on the file name and then clicking on the OK button or by double-clicking on the file name. As the File Selector dialog box only shows nine files at a time, the user may scroll through the remaining files. In addition, the user may also type in a new file name or directory.

To change the directory, the user needs only to position the mouse cursor at the directory field location that should be changed and then click. This will move the text cursor (the cross hairs) to the click position. For example, if the directory field shows "C:\pictures\*.gem" and the user wants "C:\gemapps\*.gem", the user can move the mouse cursor to the second backslash (just before *.gem), delete the string "pictures", and then type in "gemapps". The display should change to reflect the new directory selections. **fsel_input**( ) will return the full file name for the file that was selected, as well as which button was selected (OK or Cancel).

File Selector Library Syntax Summary

This section is a short reference summary for the single function in the File Selector Library.

Displaying the File Selector Dialog Box: fsel_input( )

The **fsel_input**( ) function displays the File Selector dialog box and returns the results of the user interaction.

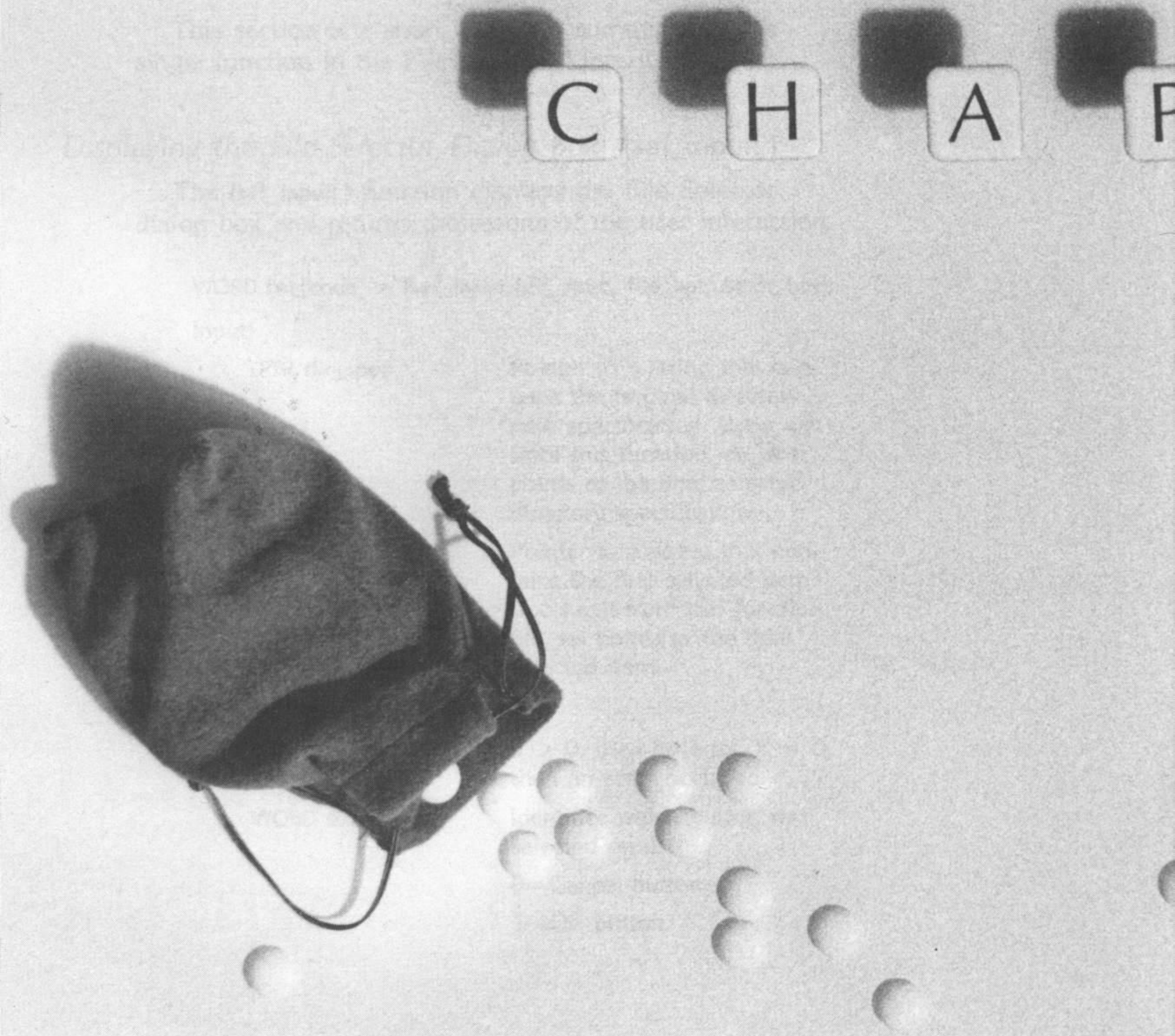
WORD ret_code = **fsel_input** (dir_spec, file_sel, &exit_but);

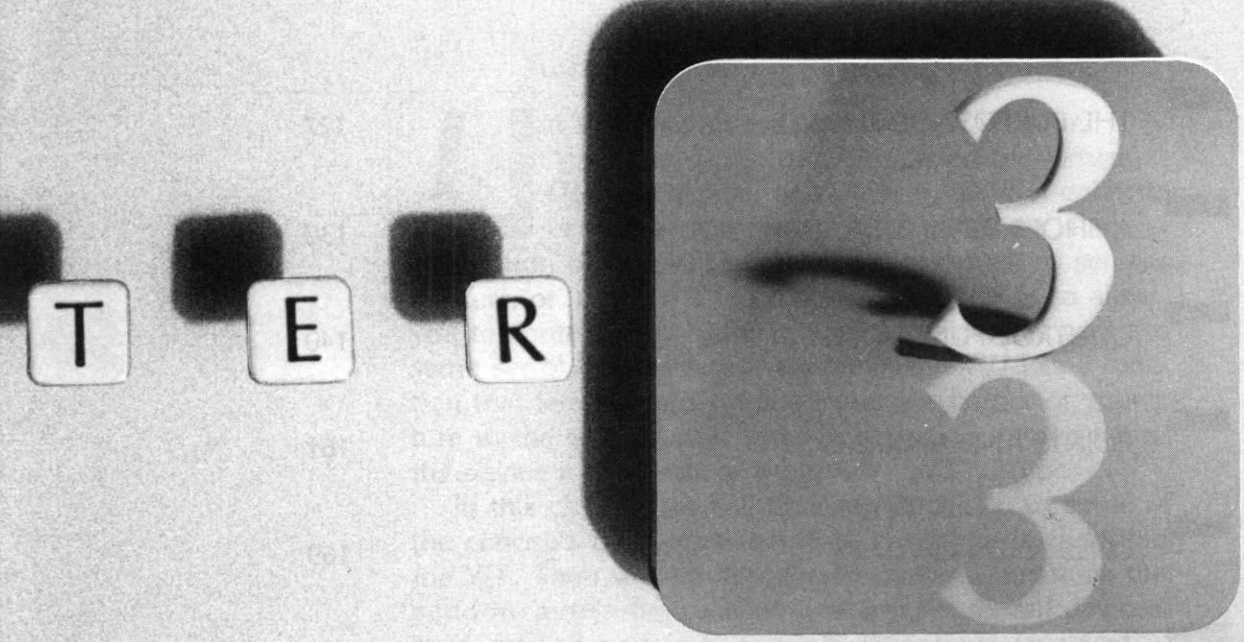
Input:

LPTR dir_spec	Pointer to a string that contains the original directory path specification. Upon exit from this function, dir_spec points at the final selected directory specification.
LPTR file_sel	Pointer to a string that contains the first selected item. Upon exit from this function, file_sel points at the final selected item.

Output:

WORD ret_code	If > 0, then no error. If = 0, then an error occurred.
WORD exit_but	Identifies which button was selected on exit: 0—Cancel button 1—OK button.


C H A P



VIRTUAL DEVICE INTERFACE— THE VDI

In this chapter we describe the "G"—for "graphics"—part of GEM, called the *Virtual Device Interface,* or *VDI.* The VDI is a collection of drawing functions. A *virtual device* is an abstraction of physical graphics devices, such as screens, printers, or plotters. The purpose of the VDI is to allow you to control many different graphical devices with the same set of functions. For example, the VDI has a function that lets you draw a line from one corner of your picture to the other corner without having to know whether the device is a screen or a printer.

In this chapter, we will first introduce you to some of the concepts and vocabulary that are useful in describing the VDI. Then we will describe the control functions that perform certain housekeeping chores for virtual devices. Next, we will explain how to output various graphical items, such as points, markers, lines, and shapes. We will also discuss how to display graphical text with the VDI. Next, the chapter will focus on the special raster operations of the VDI. Finally, we will touch on functions that will allow you to get additional device-specific information.

THE PURPOSE OF THE VDI

One of the primary goals of the designers of GEM was to make it as easy as possible for programmers to write efficient and portable graphics-based applications. To make the graphics application easier to write, the VDI provides a large number of powerful functions. To make the graphics application portable, the VDI defines an abstraction of a graphics device, which in turn allows your program to control different physical devices with the same set of functions. Finally, to make the graphics application efficient, the VDI requires that certain conventions be followed. (If these conventions are not adhered to, they can in fact cause the computer to crash.) All of these features of the VDI will be described in this chapter.

The GEM VDI can be viewed as a set of functions that perform graphical input and output with devices, which are also known as *workstations.* Examples of devices include screens (which are usually a combination of a monitor, keyboard, and a mouse), printers, plotters, and film recorders. Another kind of "device" is a *metafile,* which is a disk file containing a series of graphics commands that can be input to other GEM applications (such as GEM DRAW or OUTPUT). In Chapter 6 we recommend that your program draw to a metafile for hard-copy output instead of drawing directly to a hard-copy device like a printer. The purpose of this is to maximize the flexibility of your program's output, since metafiles can be displayed on any device using the OUTPUT utility.

Although the VDI also provides functions that accept input from keyboards, mice, graphic tablets, and other pointing devices, if your program uses any AES functions, it must use AES functions (covered in Chapter 2) for accepting input. This is because the AES installs special input handling routines into the VDI and maintains separate processes, such as the Screen Manager, which may also be accepting input.

Figure 3.1 illustrates a way of viewing the VDI. Your application program communicates with the VDI through a set of function calls, or *bindings.* These bindings load their parameters into the contrl, intin, and ptsin arrays and then cause the GEM VDI to be called with a software interrupt on the Intel 8088 processor or with a software trap on the Motorola 68000. The VDI may perform some manipulation to the information before passing it on to the workstation specified by the *workstation selector* or *device handle.* The workstation consists of a piece of code called a *device driver,* which translates the abstract VDI command into whatever device-specific actions may be necessary to produce the desired display. If any information is to be passed back to the application (for example, the device

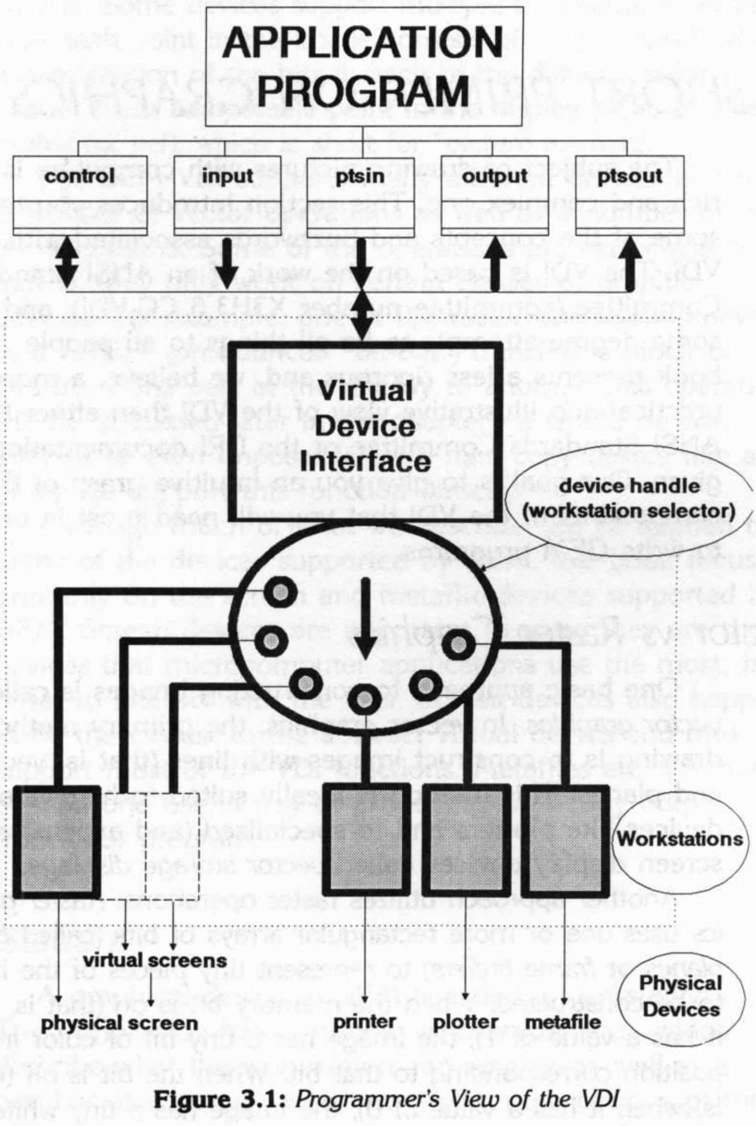


Figure 3.1: *Programmer's View of the VDI*

resolution), it is loaded into the intout and ptsout arrays and made available to the application program through the bindings.

A SHORT PRIMER ON GRAPHICS

The subject of drawing pictures with computers is a rich and complex one. This section introduces you to some of the concepts and buzzwords associated with the VDI. The VDI is based on the work of an ANSI Standards Committee (committee number X3H3.6 CG-VDI), and to some degree attempts to be all things to all people. This book presents a less rigorous and, we believe, a more practical and illustrative view of the VDI than either the ANSI Standards Committee or the DRI documentation has given. Our goal is to give you an intuitive grasp of the components of the VDI that you will need most in order to write GEM programs.

Vector vs Raster Graphics

One basic approach to constructing images is called *vector graphics.* In vector graphics, the primary method of drawing is to construct images with lines (that is, vectors) and planes. This method is ideally suited to hardware devices like plotters and to specialized (and expensive) screen display devices called *vector storage displays.*

Another approach utilizes raster operations. *Raster graphics* uses one or more rectangular arrays of bits (called *bit planes* or *frame buffers*) to represent tiny pieces of the image to be constructed. When the memory bit is on (that is, when it has a value of 1), the image has a tiny bit of color in the position corresponding to that bit. When the bit is off (that is, when it has a value of 0), the image has a tiny white spot. If there is only a single plane of color information, the image is called a *monochrome image,* and the colors displayed are called black and white (even though they may be displayed as black and green, for example, on your display screen). Some devices support multiple bit planes, in which case each point in the image consists of a color specified by a combination of the bits in each of the different color planes. Each addressable point in the display plane is called a *pixel* (or *pel*), which is short for "picture element."

The GEM VDI supports many different devices and has a number of vector operations as well as a number of raster operations. Some of the operations are *device-specific:* that is, they only work on certain classes of graphic devices. For example, one of the raster operations known as a *BITBLT* (pronounced "bit-blit") transfers a block of bits from one part of the display to another (this operation will be discussed later in the chapter). It would be very difficult or even impossible for a hard-copy device like a plotter to support this function directly.

Although much of what we describe can be applied to many of the devices supported by GEM, this book focuses primarily on the screen and metafile devices supported by GEM. Screen devices are important because they are the devices that microcomputer applications use the most, in order to interact with the user. Screen devices also happen to be the closest to the abstract virtual device and thus support most of the VDI functions. Metafiles are the most general and useful way of producing hard-copy output from your program.

Workstations

A graphical device in GEM is called a *workstation.* The workstation has a number of *characteristics,* which describe what the workstation can display, as well as a number of *attributes,* which control certain options of the display functions on that workstation. An example of characteristics is the number of addressable pixels that can be displayed vertically and horizontally. An example of attributes is the color and thickness of a line. As attributes, color and thickness can be changed for each new set of lines your program wants to display. Thus, while characteristics are fixed for each device, attributes can be changed.

In your GEM program, you deal with workstations in a manner analogous to dealing with disk files. First, you specify the desired workstation and open it, then you output display information to the workstation, and finally you close the workstation when you are finished. Most workstations can only handle one set of input and output operations. For example, if two different programs tried to display to the printer at the same time, the printer output would become very confused. GEM provides spooler programs (OUTPUT.APP and a Print Spooler desk accessory) to handle sharing devices like printers and plotters by allowing one process at a time to use the device.

The screen device, however, needs to be shared by your program, the Screen Manager, and various desk accessories all at the same time. In order to allow different programs to share the screen, the GEM VDI supports *virtual workstations*. A virtual workstation is an independent collection of device attribute values. In almost every way, a virtual workstation resembles a physical workstation, except that first, there may be more than one virtual workstation sharing and displaying information on the same physical screen workstation, and second, you use different VDI calls to open and close virtual workstations than physical workstations. Since your program has opened a virtual workstation to the screen device, your program can change any of the drawing attributes on the screen without needing to worry about other processes changing the screen attributes for their purposes.

One way to think of virtual workstations is to think about time-sharing computers that support a number of different users, each running their own program on their own terminals. Each user is controlling the computer and making it behave in a way that may be different from the way other users are making the computer behave. For instance, one user might be running an editor while another user is running a spreadsheet. In Figure 3.2, one GEM process (the Screen Manager, for example) might assume it is writing to the menu bar in a small typeface. In the figure, the Screen Manager's view of the physical workstation is represented by virtual workstation #1. At the same time, your text editor program might be displaying text in a larger typeface on virtual workstation #2. By opening its own virtual workstation onto the screen, your program can set the VDI attributes without worrying that they might be altered by other parts of GEM that also write to the screen.

WORKSTATIONS, VIRTUAL WORKSTATIONS, AND WINDOWS

It is important to be clear about the difference between a workstation and a window. A workstation *is a device, such as a screen or printer, whereas a* window *is the reservation of a portion of the device (in GEM, only the screen device has windows). The VDI provides functions that deal with workstations and virtual workstations. The AES provides functions to manage windows. GEM allows your program to display information anywhere on the workstation; your program by convention limits its display to the reserved area inside its windows.*

When your GEM program displays information in a window, what it is really doing is displaying information to a particular location on a virtual workstation, which is either set by your program or controlled by the user. The program usually models what is displayed in the window as information displayed on a piece of paper and thus gives the user the capability of choosing which part and how much of the piece of paper to look at by using the scroll bars. The real power of windows is to allow the user of your program to tailor to his or her needs the amount of information presented by your program.

When your program opens a workstation (or a virtual workstation), GEM returns a value known as a *device handle* or a *VDI handle* (we called it a device handle or "workstation

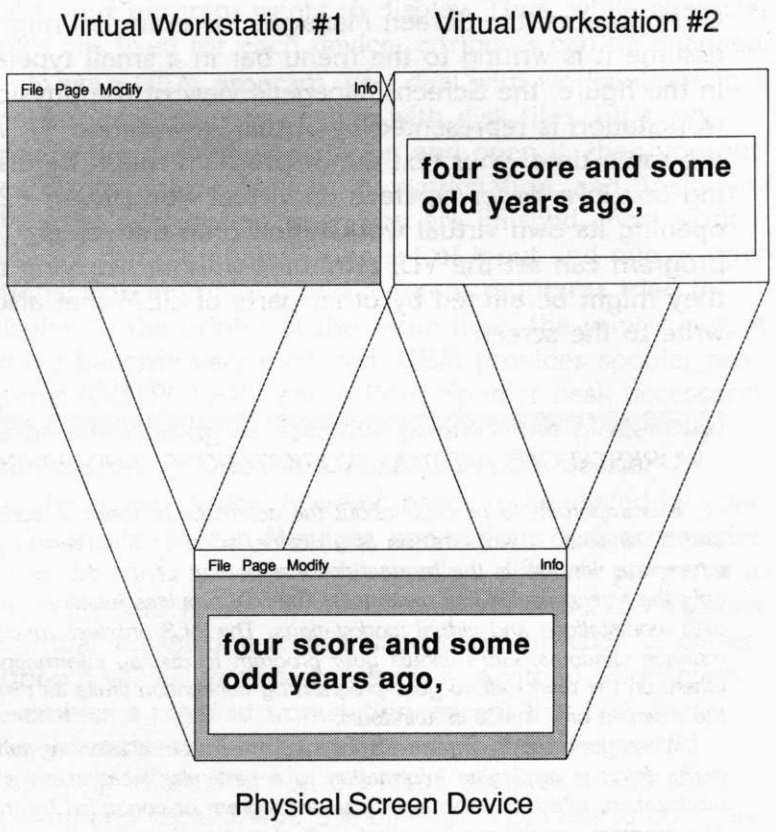


Figure 3.2: *Virtual Workstations Allow the Screen To Be Shared*

selector" in Figure 3.1). Because this handle uniquely identifies the device and its current attributes, it is a required parameter for every VDI function.

Coordinate Systems

When your program opens a workstation, GEM allows it to choose between two different methods of specifying points on the screen. The first way is called *normalized device coordinates* or *NDC.* NDC allows you to visualize the screen as a collection of points 32768 wide by 32768 high, with the origin (0,0) point in the lower left corner of the device. When you choose to use the NDC system (in the Open Workstation calls **v_opnwk( )** and **v_opnvwk( )**, discussed later in the section on "Control Functions"), the collection of all points on the screen is called *NDC space.* You can use the NDC method to make your programs simpler (with certain limitations, as we'll explain later) since the coordinate ranges are constant.

The second method is called *raster coordinates* or *RC.* With raster coordinates, the screen is presented as a collection of points with a device-specific number of points in width and height. This collection is known as *RC space.* The origin (0,0) is in the upper left corner of the screen; the X (first) coordinate specifies how many bit columns to the right, and the Y (second) coordinate specifies how many bit rows down from the top of the screen. The width and height of the screen are made available when your program opens the virtual workstation or through a call to a device information inquiry function.

At first glance, the NDC system seems more attractive because it is simpler and constant from device to device. The RC system has several advantages, however. First, all coordinates must eventually be translated into coordinates that the device can deal with. This translation is performed by the VDI when you use NDC space. Complex figures can be drawn faster in RC space, however, because the coordinate translation is unnecessary. Second, the RC system is more natural for raster operations, for specifying coordinates more precisely, and for calculating raster image

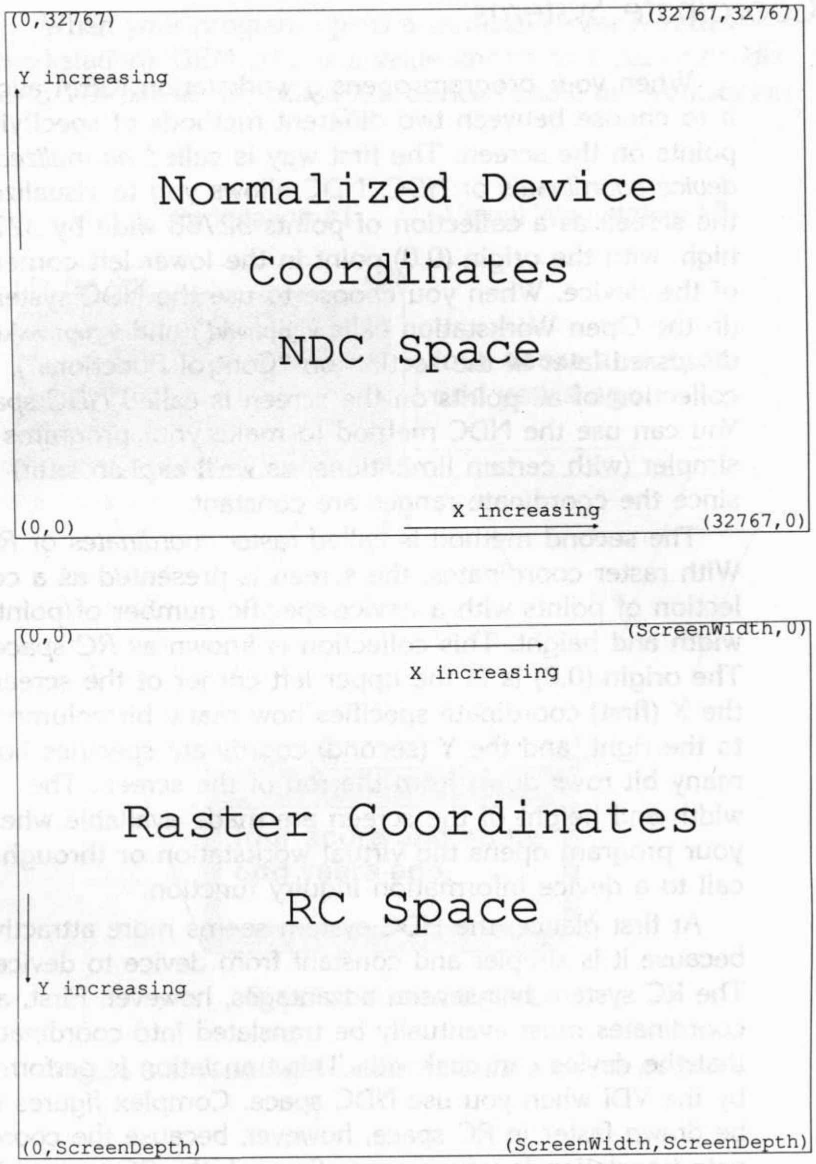


Figure 3.3: *Normalized Device Coordinates and Raster Coordinates*

memory requirements. Third, the AES uses raster coordinates exclusively in all screen references, so that if your program uses many of the AES functions, it won't be able to use the **objc_draw**( ) (Object Draw) function with NDC.

In Figure 3.3, we show the placement of different points in NDC space and RC space. As you can see by the arrows, the X coordinates increase in the right-hand direction for both coordinate systems, but the Y coordinate values increase in different directions in each system.

More sophisticated graphical applications use yet another coordinate system called *world coordinates* or *image coordinates.* World coordinates represent whatever coordinate system is most appropriate for the particular task at hand. For example, a drawing application might best use a coordinate system measured in inches (or fractions thereof). To work with the VDI, the application must then translate each point from world coordinates to either NDC or RC. In this case, RC is much more appropriate because it allows the application more control over round-off error. If the application translates first to NDC, the VDI then translates to RC, and the error accumulates.

Aspect Ratio

In high school geometry class, we learned that a circle is an object that consists of a set of points all a certain distance (called the radius) away from a single point (called the center of the circle). In GEM (and most other graphics computer systems), however, a circle is something slightly different. Put another way, if we try to draw a circle the way we learned in high school, we'll likely find ourselves drawing an ellipse.

The reason for this peculiar behavior is that in the coordinate systems and graphics devices we use in GEM, a fixed number of units in one direction is not the same length as the fixed number of units in the perpendicular direction. In NDC space, for example, the entire screen width measures 32768 units, as does the entire screen height. The only way that these units would in fact measure the same in height as they do in width would be if your display screen were square, and few common graphic display screens available today are square. The ratio of height to width is called the *aspect ratio.* Figure 3.4 illustrates the aspect ratio in NDC space.

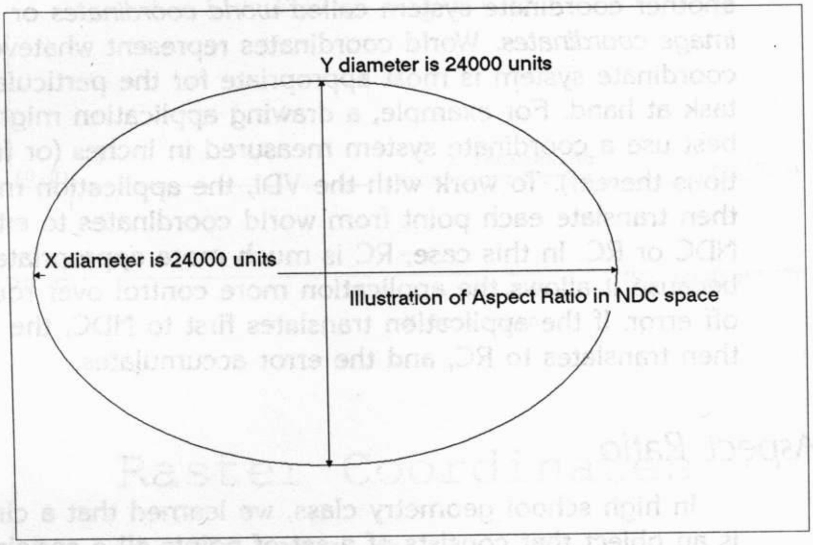

Y diameter is 24000 units
X diameter is 24000 units
Illustration of Aspect Ratio in NDC space

Figure 3.4: *Aspect Ratio in NDC Space*

As it turns out, most graphics devices have rectangular pixels instead of square pixels, which means that the aspect ratio is rarely equal to 1 in RC space either. GEM provides the pixel width and height, in microns, in the **v_opnwk**( ) and **v_opnvwk**( ) functions. This allows your program to calculate the aspect ratio in RC space, so that it can adjust the shapes of the figures it draws. Figure 3.5

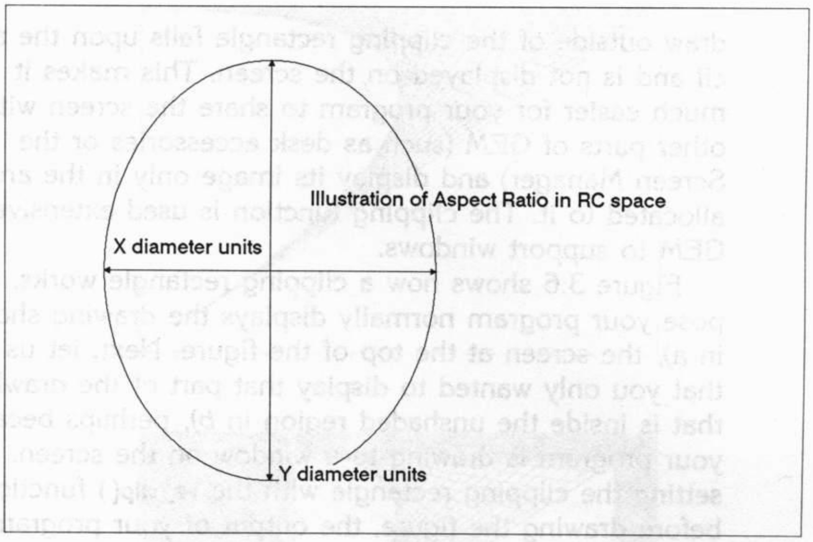


Figure 3.5: *Aspect Ratio in RC Space*

illustrates the aspect ratio in RC space for a specific device (in this case, the IBM Enhanced Graphics Adapter in monochrome mode).

Now for some good news for you programmers who want to draw circles (and parts of circles) without calculating the aspect ratio: GEM also provides some functions that take aspect ratio into account. Specifically, GEM provides *generalized drawing primitives* or *GDPs,* which draw circles and pie slices that appear circular, and not elliptical. These functions are discussed later in this chapter.

Clipping

One of the most useful functions in the VDI is the *clipping function.* This function allows you to specify a *clip rectangle,* which is like a rectangular stencil that is laid upon the display plane. Any part of the image that you draw outside of the clipping rectangle falls upon the stencil and is not displayed on the screen. This makes it much easier for your program to share the screen with other parts of GEM (such as desk accessories or the Screen Manager) and display its image only in the areas allocated to it. The clipping function is used extensively in GEM to support windows.

Figure 3.6 shows how a clipping rectangle works. Suppose your program normally displays the drawing shown in *a)*, the screen at the top of the figure. Next, let us say that you only wanted to display that part of the drawing that is inside the unshaded region in *b)*, perhaps because your program is drawing to a window on the screen. By setting the clipping rectangle with the **vs_clip()** function before drawing the figure, the output of your program will look like *c)* of Figure 3.6.

One of the conventions for efficiency that we mentioned earlier involves the clipping rectangle: your program runs faster if you leave clipping off. If you do this, however, you must be very careful that your program does not draw to coordinates outside the screen coordinates, since in doing so your program and the VDI could write over sensitive portions of memory and could cause the computer to crash. Most applications will choose to accept the performance impact in order to eliminate the need to be concerned about only drawing in certain areas.

At this point, you have been introduced to number of concepts and capabilities of the VDI. Let us now discuss the individual VDI functions, starting with the fundamental VDI workstation control functions.

CONTROL FUNCTIONS

Since your program needs to open the workstation before it can display to it, this first section contains information

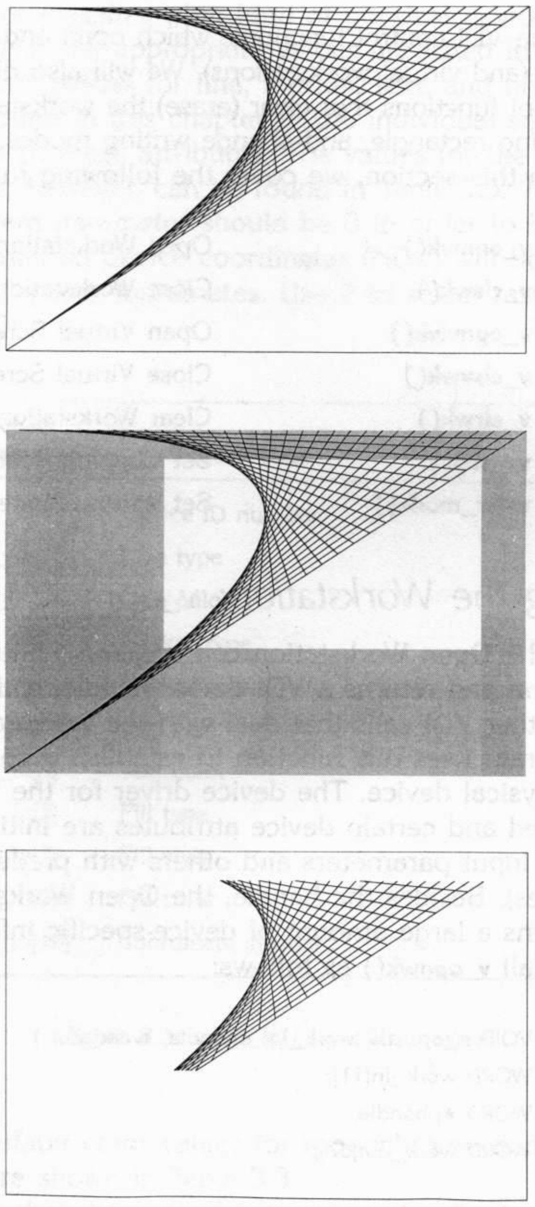


Figure 3.6: *How a Clipping Rectangle Works*

on the VDI control functions, which open and close workstations (and virtual workstations). We will also discuss other control functions that clear (erase) the workstations, set the clipping rectangle, and change writing modes.

In this section, we cover the following functions:

v_opnwk( )	Open Workstation
v_clswk( )	Close Workstation
v_opnvwk( )	Open Virtual Screen Workstation
v_clsvwk( )	Close Virtual Screen Workstation
v_clrwk( )	Clear Workstation
vs_clip( )	Set Clipping Rectangle
vswr_mode( )	Set Writing Mode

Opening the Workstation: v_opnwk( )

The Open Workstation call **v_opnwk( )** initializes a workstation and returns a VDI device handle, which is used in all other VDI calls that deal with the workstation. Your program uses this function to establish a connection with a physical device. The device driver for the workstation is loaded and certain device attributes are initialized (some with input parameters and others with predefined default values). Besides the handle, the Open Workstation call returns a large amount of device-specific information.

Call **v_opnwk( )** as follows:

```
VOID v_opnwk( work_in, phandle, work_out )
WORD work_in[11];
WORD *phandle;
WORD work_out[57];
```

The work_in array contains the desired initial attributes for the device. These are listed in Table 3.1. The Device

ID number specifies what kind of physical device is to be opened, and the appropriate values are listed in Table 3.2. Appropriate values for line, marker, text, and fill attributes can be found in this chapter in the individual sections dealing with these attributes. The values for the color attributes, however, can be found in Table 3.3. The coordinate system parameter should be 0 in order to indicate that normalized device coordinates (NDC) will be used to specify all point coordinates. Use 2 to select raster coordinates (RC).

Input Parameter	**Description**
work_in[0]	Device ID number
work_in[1]	Line type
work_in[2]	Line color
work_in[3]	Marker type
work_in[4]	Marker color
work_in[5]	Text face
work_in[6]	Text color
work_in[7]	Fill type
work_in[8]	Fill style
work_in[9]	Fill color
work_in[10]	Coordinate system

Table 3.1: *work_in[ ] Input Parameters*

The default color values for monochrome and color devices are shown in Table 3.3.

Notice that the second parameter, phandle, is a pointer to the VDI handle. This allows the **v_opnwk()** function to

Device Type	Device ID Numbers
Screen	1
Plotter	11
Printer	21
Metafile	31
Camera	41
Tablet	51

Table 3.2: *Device Identification Numbers*

return the VDI handle value. If the value assigned is 0, the **v_opnwk**( ) was unable to open the workstation. Any other value indicates that the workstation was successfully opened, and this value becomes the handle to be passed into any succeeding VDI calls on this workstation.

WHAT'S IN A HANDLE?

By now you may wonder how many different kinds of handles there are in GEM programs. The VDI handle *is used to specify which workstation to use in VDI functions. The AES has* window handles *to specify which window to use in Window Library functions. DOS has* file handles *to specify which file to use in DOS file functions. Each of these different kinds of handles are unrelated to each other, and we use the term "handle" as a generic type of identification information.*

As we can see in Table 3.4, the work_out array contains two basic types of information. The first 45 words (0–44) contain information about device characteristics. The last

Monochrome Device	**Color Indices**
0	White
1	Black

Color Device	**Color Indices**
0	White
1	Black
2	Red
3	Green
4	Blue
5	Cyan
6	Yellow
7	Magenta
8	White
9	Black
10	Dark Red
11	Dark Green
12	Dark Blue
13	Dark Cyan
14	Dark Yellow
15	Dark Cyan
16-n	Device-dependent

Table 3.3: *Color Index Values*

12 words (45–56) contain sizing information in whichever coordinate system you have chosen (in work_in[10]) for the workstation—that is, NDC or RC.

Value	Description
work_out[0]	Last addressable pixel column (X)
work_out[1]	Last addressable pixel row (Y)
work_out[2]	Precision scaling flag (0 = yes, 1 = no)
work_out[3]	Width of one pixel in microns
work_out[4]	Height of one pixel in microns
work_out[5]	Number of character heights **-1-**
work_out[6]	Number of line types
work_out[7]	Number of line widths **-1-**
work_out[8]	Number of marker types
work_out[9]	Number of marker sizes **-1-**
work_out[10]	Number of text faces
work_out[11]	Number of patterns
work_out[12]	Number of hatch styles
work_out[13]	Number of simultaneous colors (monochrome = 2) **-2-**
work_out[14]	Number of Generalized Drawing Primitives (GDPs)
work_out[15] to work_out[24]	First 10 GDPs supported **-3-**
work_out[25] to work_out[34]	Attribute set of associated GDP **-3-**
work_out[35]	Color capability flag (0 = No, 1 = Yes)
work_out[36]	Text rotation capability flag (0 = No, 1 = Yes)
work_out[37]	Fill area capability flag (0 = No, 1 = Yes)
work_out[38]	Cell array capability flag (0 = No, 1 = Yes)
work_out[39]	Number of total colors **-2-**

Value	Description
work_out[40]	Number of locator devices -3-
work_out[41]	Number of valuator devices -3-
work_out[42]	Number of choice devices -3-
work_out[43]	Number of string devices -3-
work_out[44]	Workstation type -4-
work_out[45]	Minimum character width
work_out[46]	Minimum character height
work_out[47]	Maximum character width
work_out[48]	Maximum character height
work_out[49]	Minimum visible line width
work_out[50]	0
work_out[51]	Maximum line width in X axis
work_out[52]	0
work_out[53]	Minimum marker width
work_out[54]	Minimum marker height
work_out[55]	Maximum marker width
work_out[56]	Maximum marker height

Table 3.4: *work_out[ ] Output Values*

A few words of explanation might be helpful as you look over this table. Some devices support continuous sizes for certain drawing primitives, while other devices have a set of discrete sizes for markers, line widths, and character heights (marked with **-1-** in the table). A value of 0 indicates continuous scaling is supported. The number of simultaneous colors (characteristics marked by **-2-**) supported by the device just refers to the number displayable at any given moment. Some devices have color tables (or *palettes*) that can be changed to different combinations of red, green, and blue and that can thus produce many more colors than can otherwise be displayed at any single time.

Much of the information in this array is included either for completeness or to solve a particular problem for a somewhat esoteric purpose. Those items marked by **-3-** are explained in more detail in the DRI GEM Developer's Kit and are not commonly needed by GEM applications.

Appropriate values for the workstation type (**-4-**) are as follows:

0: output only
1: input only
2: input/output
3: reserved
4: metafile

We do not recommend using the **v_opnwk**( ) function to open the screen device, because the screen device has already been opened by GEM when your program gets control. Instead, use the Open Virtual Screen Workstation function, **v_opnvwk**( ), which we will discuss in a moment.

Closing the Workstation: v_clswk( )

This function flushes any pending output to the workstation, closes it, and frees up the memory allocated to the device driver. Any further output to the device is ignored.

Call this function as follows:

```
VOID v_clswk( handle )
WORD handle;
```

Opening the Virtual Screen Workstation: v_opnvwk( )

The Open Virtual Screen Workstation is very similar to the Open Workstation a **v_opnwk**( ) call. The main differences are that **v_opnvwk**( ) only works for screen devices and that it expects handle to be the value of the physical workstation's VDI handle returned by a previous call to **v_opnwk**( ). As we mentioned in the discussion of workstations at the beginning of this chapter, virtual workstations allow several processes to share the screen. Since the Desktop opens the physical workstation first, you must get the handle of the physical screen device from the AES using the **graf_handle**( ) call. Our example in Listing 3.1 at the end of this section shows how **graf_handle**( ) is used.

Here is the procedure prologue for **v_opnvwk**( ):

```
VOID v_opnvwk( work_in, phandle, work_out )
WORD work_in[11];
WORD *phandle;
WORD work_out;
```

The WORD value pointed to by phandle must contain the handle value returned by the **graf_handle**( ) call, and it is replaced with a different handle value that identifies it as a virtual workstation. The handle is set to 0 if the virtual screen workstation cannot be opened.

Closing the Virtual Screen Workstation: v_clsvwk( )

The **v_clsvwk**( ) function signals the end of output to the virtual device, and is just like **v_clswk**( ). Any further output is ignored. The function's prototype looks like this:

```
VOID v_clsvwk( handle )
WORD handle;
```

Clearing a Workstation: v_clrwk( )

For a screen device, Clear Workstation clears the screen to white (the background color). A printer ejects the current page, a plotter ejects the current page or prompts the operator for a new sheet of paper. The function is declared as follows:

```
VOID v_clrwk( handle )
WORD handle;
```

Since the screen device is shared by more than one process in GEM, we recommend that you do not use **v_clrwk()** on the screen workstation or virtual workstations. To clear part of the screen (such as a window), use the **vr_recfl()** or **v_bar()** functions (covered in section "Rectangles and Filled Rectangles" later in this chapter).

Setting the Clipping Rectangle: vs_clip()

We discussed the purpose of the clipping rectangle in the "Short Primer on Graphics" earlier in this chapter. Clipping is very useful in that it allows your program to draw an entire figure onto the screen while clipping out everything outside of, for example, the window your program may be displaying into. Clipping is used extensively in the Demo program in Chapter 5. Although it may be expensive and less efficient in terms of compute time, it greatly simplifies your program's code to draw on the screen, especially inside windows.

The **vs_clip()** procedure prototype looks like this:

```
VOID vs_clip( handle, clip_flag, xy_array )
WORD handle;
WORD clip_flag;
WORD xy_array[4];
```

The **vs_clip()** function turns clipping on or off, depending on the value of clip_flag: 0 means turn clipping off, and nonzero means turn clipping on. You can set the clipping area as often as your program requires: that is, you can turn on clipping as often as you want, and you must turn clipping off only when you are finished drawing.

The xy_array[0] contains an X coordinate and the xy_array[1] contains a Y coordinate of a corner of the clipping rectangle. Values xy_array[2] and xy_array[3] contain the X and Y coordinates, respectively, of the opposite corner of the clipping rectangle. The coordinates are assumed to be in NDC or RC space, depending on how the workstation was opened. See Figure 3.6 for an illustration of how the clipping rectangle works.

We strongly suggest that you turn on clipping, especially in the early stages of developing your program. If you write to coordinates which are outside your screen coordinates, you can cause sensitive areas of memory to be overwritten, causing the computer to crash. The rule is: When in doubt, clip.

Setting the Writing Mode: vswr_mode( )

The *writing mode* refers to how the VDI will place the image your program is drawing onto the device. The writing mode in the GEM VDI affects almost all of the VDI functions except for the raster operations, which allow you to specify the writing mode separately.

When we draw an image with a pencil on a piece of paper, we have essentially two writing modes, corresponding to the lead and eraser ends of the pencil: the mode where we draw over whatever is on the paper, and the mode where we erase whatever is on the paper. For almost everything you draw with the VDI, the **vswr_mode( )** function gives you this capacity, and more. Given the fact that the image on the device consists of a large array of pixels, we can say that the writing mode controls how the pixels of the figure to be drawn will affect the pixels that are already displayed.

To understand the writing mode more clearly, we need to examine how each mode affects the VDI drawing functions. The call to **vswr_mode**( ) looks like this:

```
WORD vswr_mode( handle, mode )
WORD handle;
WORD mode;
```

Table 3.5 shows the different writing modes for this function.

Mode	Description
1	Replace
2	Transparent
3	XOR
4	Reverse transparent

Table 3.5: *Writing Modes*

Let's discuss each of these modes in turn, and study some examples of how the writing mode affects the VDI drawing functions at the pixel level. Figure 3.7 contains a background image and a drawing shape. In the next four figures (Figures 3.8 through 3.11), the same drawing shape is drawn on top of the background image in the same relative location in each different writing mode. The background image is assumed to be divided into two colors: black on the top and white on the bottom. The drawing shape consists of four bits, where the shaded blocks represent a bit with a 1 value and the white blocks represent a bit of value 0.

As we will find out in the remainder of this chapter, each class of shape (markers, lines, text, and so on) has

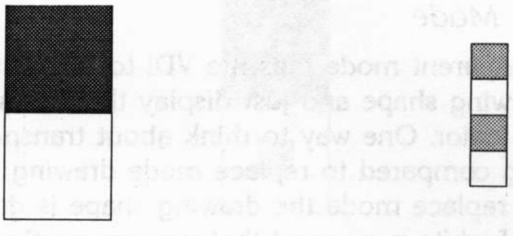


Figure 3.7: *The Background Image and the Drawing Shape*

its own separate function to set the drawing or *foreground color.* We will demonstrate how each 1 and 0 bit in the drawing shape will change the color of a pixel in the background image.

Replace Mode

This mode is the simplest to understand. When the VDI displays an image in replace mode, it writes over anything on the foreground color for each 1 bit in the drawing shape and places a pixel of value 0 (white) for each 0 bit. Figure 3.8

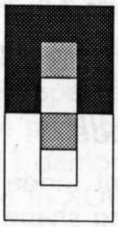


Figure 3.8: *Replace Mode*

shows what happens when the drawing shape is displayed on the background image in replace mode.

Transparent Mode

Transparent mode tells the VDI to ignore the 0 bits in the drawing shape and just display the 1 bits in the foreground color. One way to think about transparent mode drawing compared to replace mode drawing is to imagine that in replace mode the drawing shape is drawn on a piece of white paper and that you are pasting the piece of paper over whatever is already there. In transparent mode, the drawing shape is drawn on a transparent piece of plastic, and only the lines (or 1 bits) of the drawing shape affect the image.

Figure 3.9 illustrates what would result from displaying the drawing shape in transparent mode.

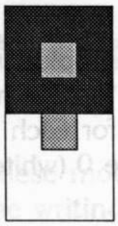


Figure 3.9: *Transparent Mode*

Reverse Transparent Mode

Reverse transparent mode only affects the image where the bits in the drawing shape are 0 (instead of where the bits are 1 as in transparent mode). Any 0 bits in the image we are drawing are changed to the current foreground color. Figure 3.10 shows the drawing shape after it has been displayed in reverse transparent mode.

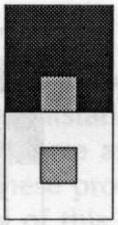


Figure 3.10: *Reverse Transparent Mode*

This writing mode is very rare, but it does have some interesting uses. You can use reverse transparent mode along with transparent mode to create two-color dashed lines by first using transparent mode to draw the line in one color and then using reverse transparent mode to draw the line in the second color. You can also create a "backdrop" color for writing text by using transparent mode, writing the text in one color, and then using reverse transparent mode and writing the text in another color.

XOR Mode

XOR mode is best for drawing something you may want to move or remove without otherwise disturbing the contents of the screen. This writing mode works by applying an exclusive OR (XOR) operation on the bits in the drawing shape and on the pixel bit values on the background image. Figure 3.11 demonstrates the result of displaying the drawing shape in XOR mode.

This mode has a very interesting property. If you display the drawing shape a second time in XOR mode, the drawing shape is cancelled out—that is, the shape is removed from the screen. The mathematical reason for this is that the XOR Boolean operation happens to be its

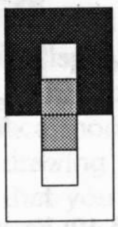


Figure 3.11: *XOR Mode*

own inverse; however, this interesting bit of knowledge is not important here. The important fact can by summed up as, "Now you see it, now you don't."

You might use this mode for displaying simple animation effects, like a movable rubber box or flashing icon. The reason this mode is used for *movable* objects is that you can

— 1 Display the object.

— 2 Display it again, thus restoring the previous image (that is, you've erased it).

— 3 Change the object (that is, make it bigger, move it, and so on).

— 4 Go to step one.

The overall effect is a very inexpensive method for turning an image on and off. One side effect of this, however, is that the effect works best on a solid background. If the background is dithered (that is, patterned in a way that produces a gray halftone), the drawing shape displayed in XOR mode may not be recognizable.

Examples of Opening a Virtual Screen Workstation

At this point, we're ready to look at some examples of opening a virtual screen workstation. Although the two procedures we will present here are different, they do almost the same thing. These procedures are used over and over again in the rest of this chapter. These two procedures open and clear a screen workstation, and then call an example procedure. In one case (ctrlndc.c), the virtual workstation is opened using normalized device coordinates (NDC). The actual display procedure, called from the

```
/* CTRLNDC.C - main driver for draw_ndc() examples */
#include "portab.h"
WORD contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

#define M_OFF 256
#define NDC_COORDS 0
#define RIGHT_ALIGNED 2
#define BOTTOM_ALIGNED 3

VOID GEMAIN()
{
   WORD handle, work_in[11], work_out[57];
   WORD ii;

   appl_init();                                  /* init AES for next call */
   handle = graf_handle( &ii,&ii,&ii,&ii );      /* get screen handle */
   graf_mouse( M_OFF, NULL );                    /* hide mouse */
   v_clrwk( handle );                            /* clear workstation */

   for( ii=0; ii<11; ++ii ) work_in[ii]=1;       /* init work_in array */
   work_in[10] = NDC_COORDS;                     /* use NDC coordinates */
   v_opnvwk( work_in, &handle, work_out );       /* open the workstation */

   draw_ndc( handle );                           /* do the example routine */

   vst_alignment( handle,RIGHT_ALIGNED,BOTTOM_ALIGNED,&ii,&ii );
   v_gtext( handle,32767,0,"Press any key to continue" );
   evnt_keybd();                                 /* pause for viewing */
   v_clsvwk( handle );                           /* close the workstation */
   appl_exit();                                  /* tell AES we're through */
}
```

Listing 3.1: *CTRLNDC.C: Main Driver for draw_ndc() Examples*

program in Listing 3.1, is called **draw_ndc()**, and the only parameter is the device handle. We use NDC coordinates to simplify some examples. In the other case (ctrlrc.c), the virtual workstation is opened using raster coordinates (RC). The example drawing procedure in Listing 3.2 is called **draw_rc()**, and it takes a handle, a starting position X and Y, a screen_width, and a screen_height. This information is required for working with windows; we use it because your program should only be writing to a portion of the screen

```
/* CTRLRC.C - main driver for draw_rc() examples */

#include "portab.h"
WORD contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

#define M_OFF 256
#define RC_COORDS 2
#define RIGHT_ALIGNED 2
#define BOTTOM_ALIGNED 3

VOID GEMAIN()
{
    WORD handle, work_in[11], work_out[57], max_w, max_h;
    WORD ii;

    appl_init();                                 /* init AES for next call */
    handle = graf_handle( &ii,&ii,&ii,&ii );     /* get screen handle */
    graf_mouse( M_OFF, NULL );                   /* hide mouse */
    v_clrwk( handle );                           /* clear workstation */

    for( ii=0; ii<10; ++ii ) work_in[ii]=1;      /* init work_in array */
    work_in[10] = RC_COORDS;                     /* use RC coordinates */
    v_opnvwk( work_in, &handle, work_out );      /* open the workstation */

    max_w = work_out[0]; max_h = work_out[1];
    draw_rc( handle,0,0,max_w,max_h );           /* do the example routine */

    vst_alignment( handle,RIGHT_ALIGNED,BOTTOM_ALIGNED,&ii,&ii );
    v_gtext( handle,max_w,max_h,"Press any key to continue" );
    evnt_keybd();                                /* pause for viewing */
    v_clsvwk( handle );                          /* close the workstation */
    appl_exit();                                 /* tell AES we're through */
}
```

Listing 3.2: *CTRLRC.C: Main Driver for draw_rc() Examples*

(the portion inside the window). In order to be able to work with windows, we recommend that you use raster coordinates for your programs.

These procedures use a number of AES calls that were documented in Chapter 2. These calls are

appl_init()	Initialize application
graf_handle()	Get screen handle
graf_mouse()	Turn mouse cursor off (or on)
evnt_keybd()	Wait for something to be typed on the keyboard
appl_exit()	Terminate application

Two of the functions, graf_mouse() and evnt_keybd(), were used to hide the cursor and to let the program pause. Your GEM program needs to hide the cursor whenever it draws anything to the screen because of the way GEM draws the cursor. Specifically, GEM saves an area of the screen before drawing the cursor, and restores it when the cursor is moved or hidden. If your program doesn't hide the cursor, then any new drawing in the area around the cursor form will be destroyed when the cursor is next moved. Note that although there are equivalent input functions in the VDI, you must use AES functions for input in order to remain consistent with the services that the AES offers, including event handling and the Screen Manager.

We used the other three AES functions because GEM starts an application with the physical screen already opened. If we tried to use the Open Workstation call **v_opnwk()**, it would have failed because the screen was already open. Thus, we called the **graf_handle()** function to ask the AES for the physical VDI handle of the screen. We used the **appl_init()** function so that the AES would be properly initialized for the **graf_handle()** call. Finally, we used the **appl_exit()** call to balance out the **appl_init()** call and release allocated resources.

What about the rest of the code in the sample procedures?

First, note the initial declaration of the arrays contrl[], intin[], ptsin[], intout[], and ptsout[]. These arrays are used by the VDI interface routines, called bindings. The main reason that the bindings require you to declare these arrays explicitly is that some of the VDI functions allow you to pass in a varying number of points or characters. Since you declare the arrays, you can make sure they are large enough to handle the largest number of input values that you plan to pass into any of the VDI routines.

Next, on line 6, the name of the main function is GEMAIN( ). GEM programs are usually linked without the rest of your C compiler's run-time library, and so we don't use the symbol main( ) in case your compiler does something special with that symbol. For more information on how a GEM program is compiled and linked together, see Chapter 6.

The **appl_init**( ) call registers the program as an AES application. This call is required before we make the second call to **graf_handle**( ) to obtain the screen VDI handle. The **graf_handle**( ) function, as we shall see in the next chapter, also returns some information about the character size. Since we do not need this information, we pass in four pointers to the same variable. The **graf_mouse**( ) call makes the mouse form disappear so that it does not obscure any of the graphics in our examples, and the **v_clrwk**( ) call sets the entire screen to white. The reason we use this function in ctrlndc.c (even though we told you not to) is that the ctrlndc.c routine assumes total control over the screen and does not share the screen with the AES menu bar or windows.

Next, we initialize the work_in[] array to all ones (1), except for work_in[10], which is set to 0 in ctrlndc.c and 2 in ctrlrc.c. This value determines the coordinate system: 0 for NDC, or 2 for RC. The reason we set all other values to 1 is that for drawing purposes, 1 is usually a more visible default. For example, 1 means black for color indices; if we used 0, we would not be able to find the white line we've just drawn on a white background. Finally, we make the Open Virtual Screen Workstation call **v_opnvwk( )**, passing in the physical screen handle and getting back the virtual screen handle. If we wanted to be very careful, we would test to make sure this value is not 0, which would mean that the call had failed.

We are almost ready to call the example procedure. In ctrlrc.c, however, we must first obtain the width and height of the screen to pass this information into the RC examples. This is unnecessary in NDC examples because we use the entire screen and, according to the definition of NDC, the screen size is always 32768 by 32768.

When the example procedure is finished, **vst_alignment( )** sets up the text alignment so that the point we specify in **v_gtext( )** is in the lower right-hand corner of the block of displayed text. For more information on what the parameters mean, see the discussion in the section called "Setting Graphics Text Alignment: **vst_alignment( )**" later in the chapter. Next, we display a nice message, and then we wait for a keyboard event, in order to allow the user some time to study the marvelous graphic. Finally, we close the virtual workstation with **v_clsvwk( )** and then use **appl_exit( )** to notify the AES that we are terminating.

POINTS AND MARKERS

Now that we have discussed the VDI functions that your program will use to control how things get drawn, it is time to talk about drawing things on the workstations. This section will cover the VDI functions that draw points (that is, dots) and markers. A *marker* is a generic term for a number of different shapes. See Table 3.6 for the different marker types supported. The type, size, and color of markers are set by calls to attribute functions. Here are the functions we will cover in this section:

vsm_type( )	Set Marker Type
vsm_height( )	Set Marker Height
vsm_color( )	Set Marker Color
v_pmarker( )	Output Polymarkers

Value	Meaning
1	Dot
2	Plus Sign
3	Asterisk
4	Square
5	Diagonal Cross
6	Diamond
7	Device-dependent

Table 3.6: *Marker Types*

Setting the Marker Type: vsm_type( )

The **vsm_type**( ) function sets up which type of marker will be displayed on subsequent calls to **v_pmarker**( ). There can be device-dependent markers, but the GEM VDI always defines at least six marker types, as shown in Table 3.6. The procedure prototype for this function looks like this:

```
WORD vsm_type( handle, marker_type )
WORD handle;
WORD marker_type;
```

If the value of marker_type is not supported on the device, the default marker type (3, an asterisk) will be used. The marker_type selected (or the default) will be returned by the function.

Setting Marker Height: vsm_height( )

The VDI allows your program to change the size of the markers drawn by **v_marker**( ) by calling the **vsm_height**( ) function. The **vsm_height**( ) procedure prototype has this form:

```
WORD vsm_height( handle, desired_height )
WORD handle;
WORD desired_height;
```

Set the desired_height to the height you want in Y axis units (NDC or RC). Remember that a distance in Y units may be different than a distance in X units because of aspect ratio (see the section on "Aspect Ratio" earlier in this chapter). In addition, you need to know that most screen devices do not support arbitrary sizes of markers, a feature that is also known as *continuous scaling.* The screen driver will, however, usually be able to draw a small number of discrete marker sizes. If the screen driver can't draw the size you requested, it selects the next smaller height it does know how to draw. The selected height is returned by the function.

If you examine Figure 3.13, which appears later in this chapter, you will note that the dot marker (type 1) never gets any bigger (look carefully, as the dots may be hard to see: they're in the top row).

Setting Marker Color: vsm_color( )

The **vsm_color**( ) function causes succeeding markers to be drawn in the color you select. If the color_index is out of range, GEM selects 1. The actual selected value is returned. See Table 3.4 for the list of default colors.

Here is the prototype for this procedure:

```
WORD vsm_color( handle, color_index )
WORD handle;
WORD color_index;
```

Drawing Polymarkers: v_pmarker()

Finally, let's talk about the function that actually causes the markers to be displayed on the workstation. The function is called **v_pmarker()**, and it looks like this:

```
VOID v_pmarker( handle, count, xy_array )
WORD handle;
WORD count;
WORD xy_array[2 * count];
```

This function displays a number (count) of markers of the type, size, and color currently selected. The term *polymarkers* just means "many markers": in effect, this function allows you to specify a number of vertices in one call. Notice that the size of the xy_array[ ] must be large enough to contain an X,Y pair of points (specified in the current coordinate system, NDC or RC) for each desired marker. The first X and Y points are in xy_array[0] and xy_array[1], respectively. The second pair of X and Y points are in xy_array[2] and xy_array[3], and so on. This function makes it easy, for example, to display just one or an entire series of markers. The X,Y pair always specifies the center of the marker.

Examples of Marker Output

To see how the marker functions might be used in a program, let's take a look at a couple of examples. Our first example, shown in Listing 3.3, demonstrates how to show a single dot. On most screens, it will be hard to find

```
/* MARKONE.C - display a single dot on the device */
#include "portab.h"
#define DOT 1
#define BLACK 1

VOID draw_ndc( handle )
WORD handle;
{
   WORD xy_array[2];

   xy_array[0] = 32767/2;                        /* x coordinate */
   xy_array[1] = 32767/2;                        /* y coordinate */
   vsm_type( handle, DOT );                      /* set marker type */
   vsm_color( handle, BLACK );                   /* set marker color */
   v_pmarker( handle, 1, xy_array );             /* output polymarker */
}
```

Listing 3.3: *MARKONE.C: Display a Single Dot*

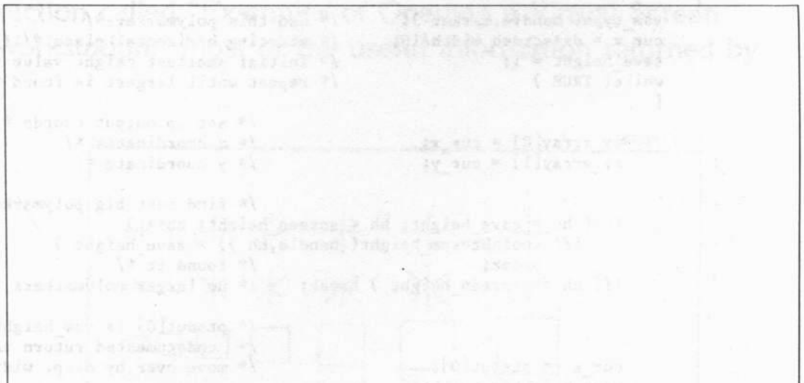


Figure 3.12: *MARKONE.C Output: Display of a Single Dot*

the dot unless you know where to look. In our example, it is in the center. The output of the program is shown in Figure 3.12.

Note that the procedure GEMAIN( ), which was presented earlier in the chapter (in the section entitled "Examples of Opening a Virtual Screen Workstation"), calls the procedure draw_ndc( ) in Listing 3.3. handle contains the device handle returned by **v_opnvwk( )**. If you want to duplicate

```
/* MARKPLY1.C - display polymarkers 1-6 using search technique */
/*          Note - this technique won't work on metafiles */

#include "portab.h"

VOID draw_rc( handle, dx, dy, screen_width, screen_height )
WORD handle, dx, dy, screen_width, screen_height;
{
    WORD xy_array[4];
    UWORD cur_x, cur_y, height, save_height, hh;
    WORD marker;
    extern WORD ptsout[];

    cur_y = dy+screen_height/16;          /* starting vertical place */
    for( marker = 1; marker < 7; ++marker ) /* for each marker type */
    {
        vsm_type( handle,marker );        /* use this polymarker */
        cur_x = dx+screen_width/16;       /* starting horizontal place */
        save_height = 1;                  /* initial smallest height value */
        while( TRUE )                     /* repeat until largest is found */
        {
                                                  /* set up output coords */
            xy_array[0] = cur_x;                  /* x coordinates */
            xy_array[1] = cur_y;                  /* y coordinate */

                                                  /* find next big polymarker */
            for( hh = save_height; hh < screen_height; hh++ )
                if( (height=vsm_height( handle,hh )) > save_height )
                    break;                        /* found it */
            if( hh >= screen_height ) break;      /* no larger polymarkers */

                                                  /* ptsout[0] is vsm_height's*/
                                                  /*  undocumented return val */
            cur_x += ptsout[0];                   /* move over by disp. width */
            save_height = height;

            v_pmarker( handle, 1, xy_array );   /* output polymarker */
        }                                 /* end line display loop */
        cur_y += 2+(screen_height/7);     /* minimum distance between rows */
    }                                     /* end marker display loop */
}
```

Listing 3.4: *MARKPLY1.C: Display Markers 1 through 6 Using Search Technique*

the examples on your computer as you proceed through the chapter, please note that we use one or the other of the different versions of GEMAIN( ) given in that earlier section in all of the remaining VDI examples.

The next example, shown in Listing 3.4, prints each different marker in as many sizes as will fit on the screen. This procedure assumes that the device only handles a discrete number of marker sizes. It uses raster coordinates because there are fewer raster coordinates than normalized device coordinates, which means that the algorithm has fewer sizes to search through. To find out what the program draws, see Figure 3.13.

The search algorithm for a new marker size is very simple: it keeps calling **vsm_height**( ) until the height returned is different from the previous height. The procedure uses the marker width (returned in ptsout[0]) to move the display of the next marker enough to avoid overwriting the previous marker.

The global ptsout[ ] array was discussed in an earlier section called "Examples of Opening a Virtual Screen Workstation." Most of the useful information returned by

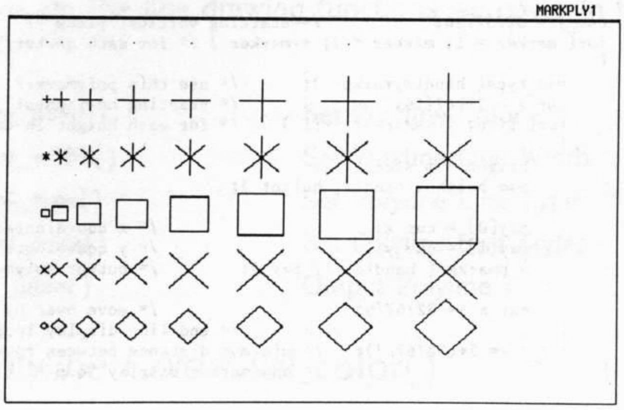


Figure 3.13: *MARKPLY1.C Output: Display Markers 1 through 6*

the VDI is passed back by the bindings either directly by the function, or, if more than one value is to be returned, by calling the functions with pointers to the information. This is the only instance where we felt that we needed some additional information from the VDI that was not passed back through the bindings.

The next program, shown in Listing 3.5, works in NDC space and draws many markers whose sizes are contained in a table of marker height values. The values in the table were achieved by trial and error, and they display different sizes of markers on different devices. You may need to play

```
/* MARKPLY2.C - display polymarkers 1-6 from a table, using NDC */
/*            Note - this technique is device-specific; the table values */
/*                 were calculated by trial and error, for EGA mono card*/

#include "portab.h"

#define NMarks 6
WORD m_sizes[NMarks] = { 500, 1000, 2000, 3000, 5000, 6000 };

VOID draw_ndc( handle )
WORD handle;
{
    WORD pxy[4];
    UWORD cur_x, cur_y, height;
    WORD marker;
    WORD ii;

    cur_y = 32767/16;              /* starting vertical place */
    for( marker = 1; marker < 7; ++marker ) /* for each marker type */
    {
        vsm_type( handle,marker );       /* use this polymarker */
        cur_x = 32767/16;                /* starting horizontal place */
        for( ii=0; ii<NMarks; ++ii )     /* for each height in table */
        {
            height = m_sizes[ii];
            vsm_height( handle, height );

            pxy[0] = cur_x;                        /* x coordinates */
            pxy[1] = cur_y;                        /* y coordinate */
            v_pmarker( handle, 1, pxy );           /* output polymarker */

            cur_x += 32767/9;                      /* move over by disp. width */
        }                             /* end line display loop */
        cur_y += 2+(32767/7);   /* minimum distance between rows */
    }                           /* end marker display loop */
}
```

Listing 3.5: *MARKPLY2.C: Display Markers 1 through 6 Using Table*

around with the values in the table in order to display different sized markers on your system.

The main reason we defined a table of values in this program is that **vsm_height()** cannot return accurate information about marker heights in metafile devices. Thus, the program listed in Listing 3.5 could also output to a metafile, whereas the program in Listing 3.4 could not. The output of the program in Listing 3.5 is very similar to that shown in Figure 3.13.

LINES AND POLYLINES

Up to this point, we have only told you how to draw markers. In this section, we will talk about drawing lines. The GEM VDI has a generalized line output function, **v_pline()**, that allows you to connect a series of points with line segments and that also controls a number of attributes of those line segments. This allows you to specify a complex shape as a series of points and to display the shape in a single call.

Here are the line drawing functions we cover in this section:

vsl_color()	Set Polyline Color
vsl_width()	Set Polyline Line Width
vsl_type()	Set Polyline Line Type
vsl_ends()	Set Polyline End Styles
v_pline()	Output Polyline

Setting Polyline Color: vsl_color()

The functions for setting line attributes are analogous to the VDI functions for setting marker attributes. The function to set the color of the line to be displayed, **vsl_color**( ), is almost exactly the same as its counterparts for markers and, as we will see later in this chapter, for text and fill areas.

The **vsl_color**( ) procedure prototype looks like this:

```
WORD vsl_color( handle, color_index )
WORD handle;
WORD color_index;
```

This function makes your lines colorful. Consult Table 3.4 for the values for color_index.

Setting the Line Width of Polylines: vsl_width( )

In our discussion of drawing markers, we explained how the VDI allows your program to set the height of markers. The VDI also allows your program to change the width (or thickness) of the lines it draws. The function to set the line width, **vsl_width**( ), has a prototype that looks like this:

```
WORD vsl_width( handle, line_width )
WORD handle;
WORD line_width;
```

The line_width is in X units of magnitude (NDC or RC), which is important when you are dealing with aspect ratio. This is why the function is called **vsl_width**( ) instead of vsl_height. Note that while the VDI specifies text and marker sizes by height, lines are specified by width.

Since many devices do not continuously scale the line widths, the VDI provides the number of available line widths to your program in work_out[7] when your program opens the workstation (see Table 3.3). The actual line_width that gets set is returned by the function. It equals the closest line width supported by the device driver that is less than or equal to the requested line_width. The exception here is when you try to set line_width to a size smaller than a pixel (such as 0), whereupon the VDI uses a width of at least one pixel.

For a sample of **vsl_width**( ) in action, see linestyl.c in Listing 3.6. The output is shown in Figure 3.14.

```
/* LINESTYL.C - display various polyline widths and styles */

#include "portab.h"
#define SQUARED 0
#define ARROWED 1
#define ROUNDED 2

WORD thick_tab[4] = { 1, 4, 8, 16 };      /* use multiples of these widths */

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD pxy[4], cur_x, cur_y, incr_x, incr_y, sep_y;
    WORD tt, thickness, begin_style, end_style;

    thickness = swidth / 600;             /* magic multiplier: system dependnt*/
    for( tt = 0; tt < 4; ++tt )
        thick_tab[tt] *= thickness;

    incr_x = swidth / 10;                 /* column width */
    incr_y = sheight / 4;                 /* row height */
    sep_y = incr_y / 6;                   /* distance to separate rows */
    cur_y = dy + sheight - sep_y / 2;     /* display at bottom of screen */

    for( tt = 0; tt < 4; tt++ )
    {
        thickness = thick_tab[tt];
        vsl_width( handle, thickness );
        cur_x = dx + incr_x / 2;
        for( end_style = SQUARED; end_style <= ROUNDED; end_style++ )
            for( begin_style=SQUARED; begin_style <= ROUNDED; begin_style++)
            {
                vsl_ends( handle, begin_style, end_style );
                pxy[0] = cur_x;
                pxy[1] = cur_y;
                cur_x += incr_x;
                pxy[2] = cur_x;
                pxy[3] = cur_y - (incr_y-sep_y);
                v_pline( handle, 2, pxy );
            }
        cur_y -= incr_y;
    }
    vsl_width( handle, 1 );               /* reset line thickness */
    vsl_ends( handle, SQUARED, SQUARED );
}
```

Listing 3.6: *LINESTYL.C: Sketch Various Line Widths and Styles*

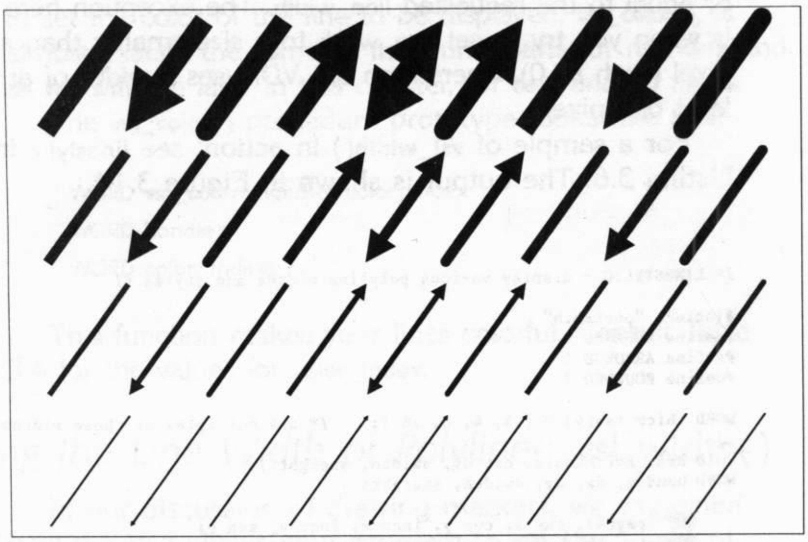


Figure 3.14: *LINESTYL.C Output: Various Line Widths and Styles*

Setting the Line Type of Polylines: vsl_type()

The VDI displays solid lines in any size. For some line widths (usually only the single-pixel width), the VDI provides several different types of line patterns. These line types can be set with the **vsl_type**( ) function, whose prototype looks like this:

```
WORD vsl_type( handle, line_type )
WORD handle;
WORD line_type;
```

See Table 3.7 for the different types of lines your program can specify. If the line_type value is out of range, it defaults to a solid line. Furthermore, if you set the width (with **vsl_width**( )) to a thicker line, the line_type value may be reset to solid.

		--- 16 bits ---	
Type	**Name**	**MSB**	**LSB**
1	solid	****************	
2	long dash	************____	
3	dot	***_____***_____	
4	dash, dot	******____***___	
5	dash	********________	
6	dash, dot, dot	****___**__**___	
7	user-defined		
8–n	device-dependent		

Table 3.7: *line_type Values*

In Table 3.7, the asterisks represent "on" bits (with value 1) and the underscores represent "off" bits (with value 0). When a line_type is selected from one of these combinations, the 16-bit pattern shown in this table is replicated over and over again along the length of the line, starting with the most significant bit (MSB). Thus, if your program specifies a line drawn in line type 6 (dash, dot, dot), for example, the bit pattern for the line will look like this:

```
111100011001100011110001100110001111000110011000
+_______________+_______________+_______________
```

The ruler line with plus and minus signs serves to delimit the start of each word.

We used the asterisk and underscore convention in Table 3.7 to try to illustrate the results more clearly. For example, the above example will look as follows:

```
****___**__**___****___**__**___****___**__**___
+_______________+_______________+_______________
```

```
/* LINETYPE.C - displays line types 1-6 */

#include "portab.h"
BYTE *labels[] = { " 0", " 1", " 2", " 3", " 4", " 5", " 6" };

VOID draw_rc ( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD pxy[4], cur_y, incr_y, line_type;

    incr_y = sheight / 7;
    cur_y = dy + incr_y;
    pxy[2] = dx + swidth / 8;
    pxy[0] = dx + 6 * (swidth / 8);

    for( line_type = 1; line_type < 7; ++line_type )
    {
        vsl_type( handle, line_type );
        pxy[1] = pxy[3] = cur_y;
        v_pline( handle, 2, pxy );
        v_gtext( handle, pxy[0], pxy[1], labels[line_type] );
        cur_y += incr_y;
    }
    vsl_type( handle, 1 );                /* reset to default */
}
```

Listing 3.7: *LINETYPE.C: Draw line_type 1 through 6*

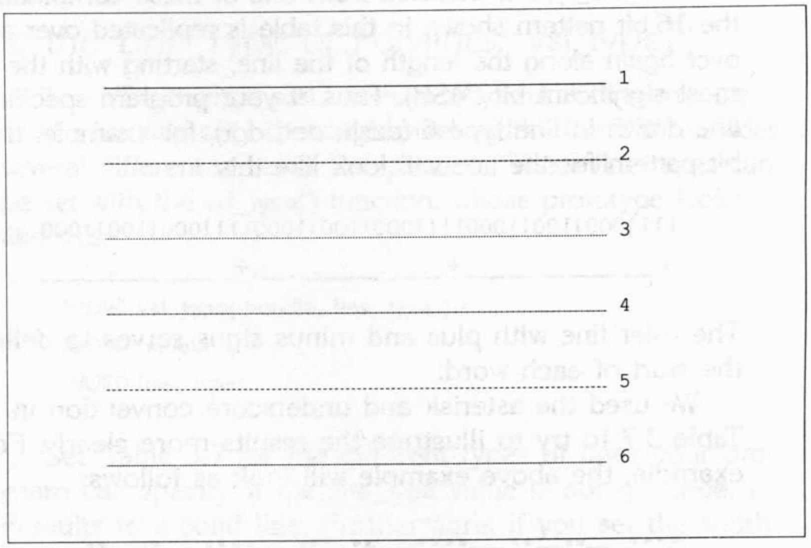


Figure 3.15: *LINETYPE.C Output: line_type 1 through 6*

User-defined line styles and device-dependent line styles are not as commonly used as the other line types. See the DRI Developer's Kit documentation for more information on these line types.

For an example of what these line types look like, see the program in Listing 3.7, which draws the first six types. The output of the program is shown in Figure 3.15.

Setting Polyline End Styles: vsl_ends( )

Another line attribute supported by the VDI is called *end styles.* End styles come in three different flavors:

0 Square (default)
1 Arrow
2 Rounded

The function **vsl_ends**( ), which is responsible for giving your program prettier line beginnings and endings, is called as follows:

```
VOID vsl_ends( handle, beg_style, end_style )
WORD handle;
WORD beg_style;
WORD end_style;
```

See Listing 3.6 for a sample of **vsl_ends**( ) in use. The output of this program is shown in Figure 3.14.

Drawing Polylines: v_pline( )

Having discussed the line attributes, we are now ready to consider the function that actually draws the lines. This function, **v_pline**( ), has the following prototype:

```
VOID v_pline( handle, count, xy_array )
WORD handle;
WORD count;
WORD xy_array[ 2 * count ];
```

Your program uses **v_pline**( ) to display simple line segments by setting xy_array[0] and xy_array[1] to the X and Y coordinates, respectively, of the starting point, by setting xy_array[2] and xy_array[3] to the X and Y coordinates of the end point of the line segment, and by setting count to 2. More complex shapes can be defined by specifying each of the polygon's vertices. (Don't forget to specify the starting point again at the end, if you want to close the polygon.) As in **v_pmarker**( ), you fill the xy_array with pairs of points, and set count to the total number of pairs. Use the various line attribute functions discussed earlier in this section before calling **v_pline**( ) to vary the color, width, type, and end styles of the line your program wants to draw.

```
/* LINEDIAG.C - draw a single diagonal line */

#include "portab.h"

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD pxy[4];

    pxy[0] = dx;                        /* x coordinates */
    pxy[1] = dy;                        /* y coordinate */
    pxy[2] = dx+swidth;
    pxy[3] = dy+sheight;
    v_pline(handle, 2, pxy);            /* output polyline */
}
```

Listing 3.8: *LINEDIAG.C: Draw a Diagonal Line*

Here are a few examples. The first example, Listing 3.8, draws a line diagonally across the screen. Figure 3.16 shows the output.

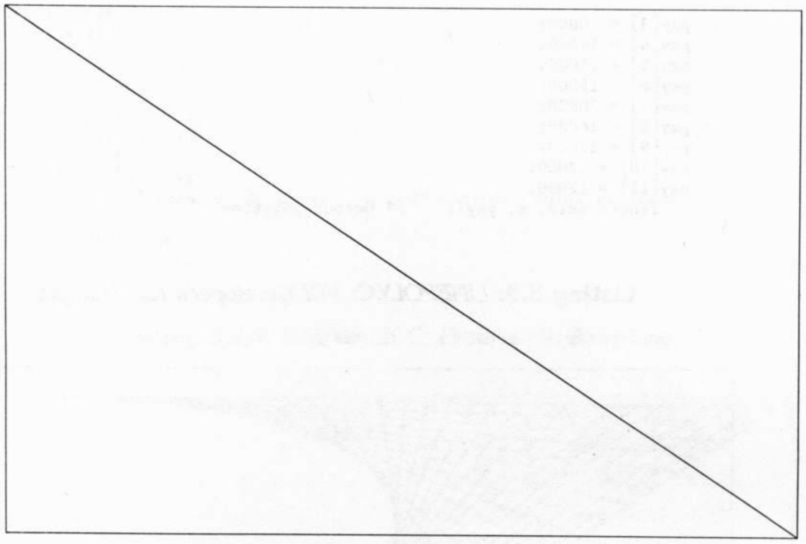


Figure 3.16: *LINEDIAG.C Output: Diagonal Line*

Next, let's display a polygon, as in Listing 3.9. The output is shown in Figure 3.17. Note that this example is very similar to the example given in the VDI section of the DRI Developer's Kit, although our example has been modified to work with the AES.

Still pretty boring. Let's draw something a bit fancier through the code shown in Listing 3.10 (its output is shown in Figure 3.18). Notice that it doesn't take many VDI commands to draw interesting shapes.

```
/* LINEPOLY.C - draw the polygon from VDI Toolkit Example, Chapter 2 */

#include "portab.h"

VOID draw_ndc(handle)
WORD handle;
{
    WORD pxy[12];

    pxy[0] = 12000;
    pxy[1] = 12000;
    pxy[2] = 12000;
    pxy[3] = 20000;
    pxy[4] = 14000;
    pxy[5] = 21000;
    pxy[6] = 16000;
    pxy[7] = 20000;
    pxy[8] = 16000;
    pxy[9] = 12000;
    pxy[10] = 12000;
    pxy[11] = 12000;
    v_pline(handle, 6, pxy);     /* Output polyline */
}
```

Listing 3.9: *LINEPOLY.C: VDI Developer's Kit Example*

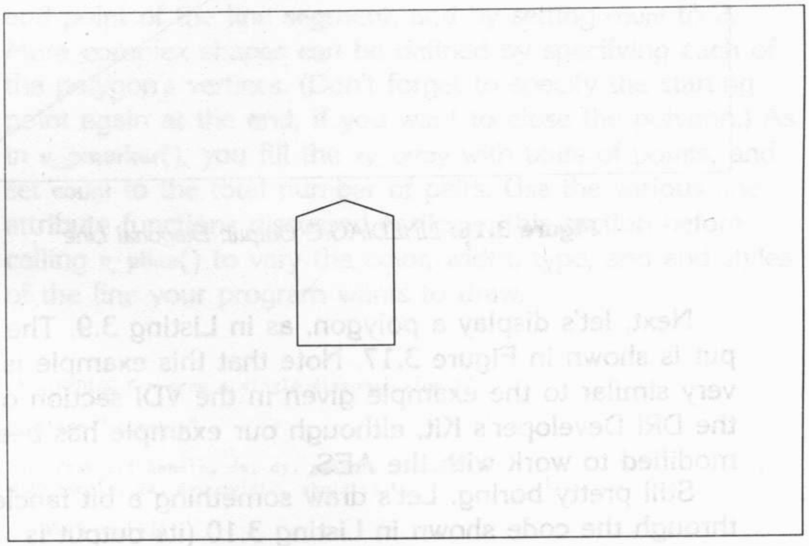


Figure 3.17: *LINEPOLY.C Output: VDI Developer's Kit Example*

```
/* LINEWALK.C - displays a "walking line" */

#include "portab.h"

VOID draw_ndc(handle)
WORD handle;
{
    WORD pxy[4];
    UWORD x1, y1, x2, y2, incr;

    x1=0;
    y1=0;
```

```
          x2=32767;
          y2=32767;
          incr = 1000;
          while( x1 < 32767 )
          {
              pxy[0] = x1;
              pxy[1] = y1;
              pxy[2] = x2;
              pxy[3] = y2;
              v_pline(handle, 2, pxy);      /* Output 2 point polyline */
              x1 += incr;
              y1 += incr;
              x2 -= incr;
          }
      }
```

Listing 3.10: *LINEWALK.C: Draw a "Walking Line"*

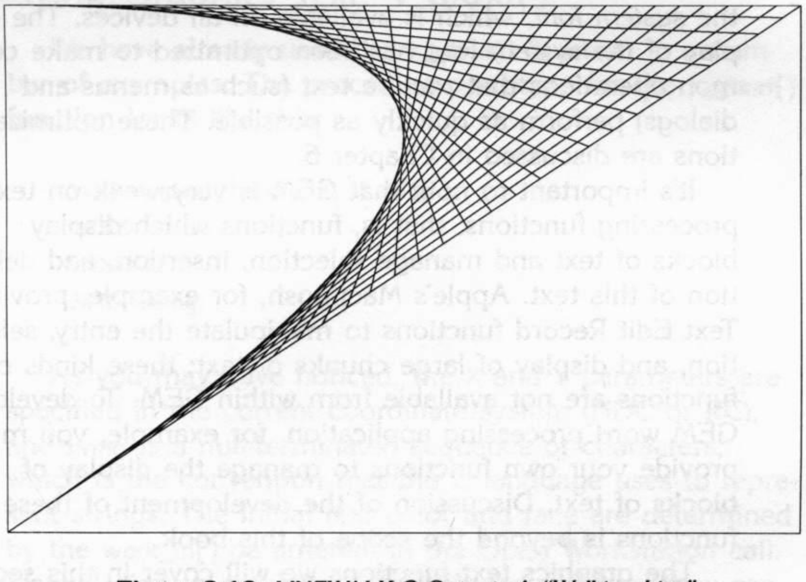


Figure 3.18: *LINEWALK.C Output: A "Walking Line"*

GRAPHICS TEXT

GEM offers two levels of functions for displaying graphics text on devices. The first level consists of the VDI functions, many of which are described in this section. The second level of text functions can be found in the Forms Library of the AES, which makes it easier to construct dialogs (that is, screen forms) for information display and input.

The VDI graphics text functions give you a lot of control over the appearance of the text that your program displays. You can control the font type, size, color, alignment, and the rotation of the text. These functions were designed to be portable between many different devices.

GEM provides a number of different styles of type, called *fonts* or *faces.* The lowest common denominator is the *system font,* which is available on all devices. The display of the system font has been optimized to make common operations that require text (such as menus and dialogs) perform as quickly as possible. These optimizations are discussed in Chapter 6.

It's important to note that GEM is very weak on text processing functions: that is, functions which display blocks of text and manage selection, insertion, and deletion of this text. Apple's Macintosh, for example, provides Text Edit Record functions to manipulate the entry, selection, and display of large chunks of text; these kinds of functions are not available from within GEM. To develop a GEM word processing application, for example, you must provide your own functions to manage the display of blocks of text. Discussion of the development of these functions is beyond the scope of this book.

The graphics text functions we will cover in this section include

v_gtext()	Output Text
vst_color()	Set Graphics Text Color
vst_height()	Set Character Height, Absolute Mode
vst_point()	Set Character Cell Height, Points Mode

vqt_extent( )	Inquire Text Extent
vst_alignment( )	Set Graphics Text Alignment
vst_effects( )	Set Graphics Text Special Effects
v_justified( )	Output Justified Graphics Text
vst_load_fonts( )	Load Fonts
vst_font( )	Set Text Face
vst_unload_fonts( )	Unload Text Fonts

Displaying Graphics Text: v_gtext( )

We have already used the **v_gtext( )** function in a number of examples. The procedure prototype for the **v_gtext( )** function looks like this:

```
VOID v_gtext( handle, X, Y, string )
WORD handle;
WORD X, Y;
BYTE *string;
```

As you may have noticed, the X and Y parameters are specified in the current coordinate system (NDC or RC), and string is a null-terminated sequence of characters, which is the convention that the C language uses to represent strings. The initial text color and face are determined by the work_in[ ] parameters in the Open Workstation call. We will discuss graphics text attributes and how they can be changed in the next few pages.

Setting Graphics Text Color: vst_color( )

Just as GEM provides functions for setting markers and lines (which we've covered in previous sections) and fill regions (which are discussed in the next section), so it also provides a function that allows your program to set the color of the graphics text displayed by your program. The procedure prototype for this function, **vst_color**( ), is as follows:

```
WORD vst_color( handle, color_index )
WORD handle;
WORD color_index;
```

See Table 3.4 for a list of colors.

Setting Character Height: vst_height( ) *and* vst_points( )

The next thing to understand is how the VDI deals with character sizes—that is, with how much room characters take up on the screen or with how big the letters are. Before we can do this, however, we need to discuss how graphics text is measured.

To get a sense of this, envision each character being displayed in its own *character cell.* The character cell has a line called the *baseline* running somewhere across the lower half of the cell. The base part of the letters rest on this line (except for the *descenders,* the bottom parts of lowercase letters like "j" and "p", which drop below the baseline). The character is centered horizontally within the character cell to give enough space around the edges for readability. Figure 3.19 illustrates the various aspects of a character's placement in a character cell.

The VDI has two functions that allow you to vary the height (and thus change the size) of the characters you display: **vst_height**( ) and **vst_point**( ). The **vst_height**( ) function uses device coordinates (either NDC or RC), while the **vst_point**( ) function allows you to specify character heights in *points.* (One point is 1/72 inch; thus, 36-point type is one-half inch high.) The **vst_height**( ) function is used to set character

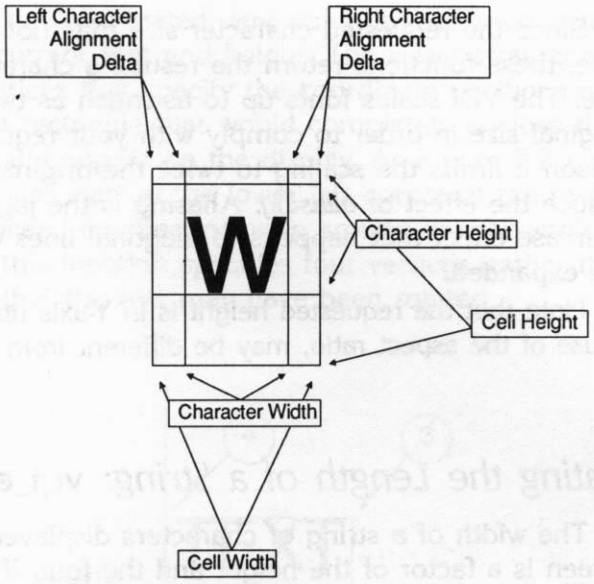

Left Character Alignment Delta
Right Character Alignment Delta
W
Character Height
Cell Height
Character Width
Cell Width

Figure 3.19: *The Character Cell*

height in *absolute mode*—that is, in NDC or RC coordinates. The **vst_point**( ) function operates in *points mode.*

The prototypes for these two functions look very similar:

```
VOID vst_height( handle, coord_height, pchar_width, pchar_height,
                 pcell_width, pcell_height )
WORD handle;
WORD coord_height;
WORD *pchar_width, *pchar_height;
WORD *pcell_width, *pcell_height;

VOID vst_points( handle, point_height, pchar_width, pchar_height,
                 pcell_width, pcell_height )
WORD handle;
WORD point_height;
WORD *pchar_width, *pchar_height;
WORD *pcell_width, *pcell_height;
```

Since the requested character size may not be available, these functions return the resulting character cell size. The VDI scales fonts up to as much as twice their original size in order to comply with your request. The reason it limits the scaling to twice the original size is to reduce the effect of *aliasing.* Aliasing is the jagged or staircase effect that happens to diagonal lines when they are expanded.

Note that the requested height is in Y-axis units, which, because of the aspect ratio, may be different from X-axis units.

Calculating the Length of a String: vqt_extent( )

The width of a string of characters displayed on the screen is a factor of the height and the font. If the font is *monospaced,* each character cell has the same width. This makes it easy to line up columns of text, for example, or to know exactly where the fifth character in a string is displayed.

Note that the system font is monospaced. Many fonts, however, are *proportionally spaced,* which means that, for example, the character cell for an "m" takes up more width than an "i". This makes text look nicer, but it makes it harder to calculate how much text fits on a display line (among other things). The VDI uses the term *text extent* to mean the width of a string displayed in the current font and height, and it provides the function **vqt_extent**( ) (Inquire Text Extent) to calculate the text extent value. The procedure prototype for **vqt_extent**( ) looks like this:

```
VOID vqt_extent( handle, string, extent )
WORD handle;
BYTE *string;
WORD extent[8];
```

The null-terminated string value is measured according to the current font and height. The extent array contains four vertices that specify the coordinate positions of the smallest rectangle that would completely enclose the string as it would appear on the display. As Figure 3.20 shows, the vertices start at the lower left corner of the rectangle and proceed counterclockwise around the rectangle. The reason this function specifies four vertices (rather than two) is that the text may have been rotated.

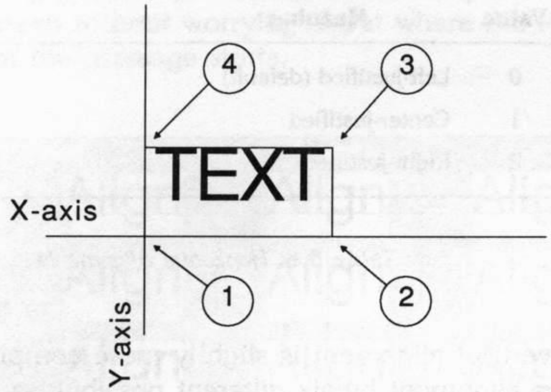


Figure 3.20: *Points Returned in extent[ ] Array*

Setting Graphics Text Alignment: vst_alignment( )

Another important factor in graphics text display is text alignment. The VDI prints out text strings relative to a given point. The relationship to that point is called the *alignment,* and it comes in two basic varieties: horizontal alignment and vertical alignment. The function to set the graphics text alignment is called **vst_alignment( )**, and its prototype is as follows:

```
VOID vst_alignment( handle, hor_in, vert_in, phor_out, pvert_out )
WORD handle;
WORD hor_in, vert_in;
WORD *phor_out, *pvert_out;
```

Horizontal alignment can be specified as shown in Table 3.8:

Value	Meaning
0	Left-justified (default)
1	Center-justified
2	Right-justified

Table 3.8: *Horizontal Alignments*

The vertical alignment is slightly more complicated, as it allows alignment by six different possibilities, as shown in Table 3.9:

Value	Meaning
0	Baseline (default)
1	Half line
2	Ascent line
3	Bottom line
4	Descent line
5	Top line

Table 3.9: *Vertical Alignments*

Note that not all alignments are available for every device. Therefore, the **vst_alignment**( ) function returns the final alignment chosen into the variables pointed to by phor_out and pver_out.

Text alignment gives you a lot of flexibility in how you can label objects on the screen. We have already demonstrated a use of **vst_alignment**( ), in the example driver code in Listing 3.1. In that example, we wanted to display a message in the lower right-hand corner of the screen. By setting the alignment to right and bottom, we can print the message to the point at the lower right-hand corner of the screen without worrying about where the left-hand portion of the message starts.

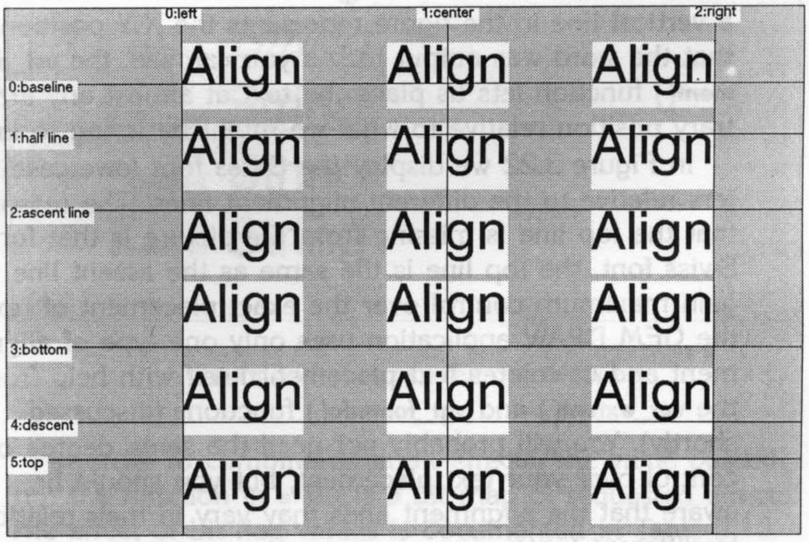


Figure 3.21: *How Text Alignment Works*

In Figure 3.21, we display a single word, "Align", at all possible alignments. Each intersection of a horizontal and

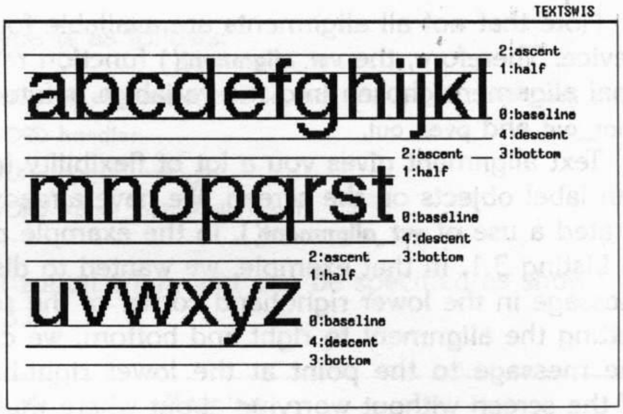


Figure 3.22: *Text Alignment Lines for Swiss Font*

a vertical line in the figure represents the X,Y position that the word was output to. As you can see, the **vst_alignment()** function lets us place the text at almost any arbitrary position relative to what we might be trying to label.

In Figure 3.22 we display the Swiss font lowercase letters relative to the different alignment lines. The reason that the top line is missing from the picture is that for the Swiss font, the top line is the same as the ascent line. To gain maximum control over the exact placement of text, the GEM DRAW application uses only one type of alignment and calculates text placement itself with help from the **vqt_extent()** and **vqt_fontinfo()** functions (discussed shortly). You will probably not need the same degree of control over your text placement, but you should be aware that the alignment lines may vary in their relation to different fonts.

Setting Special Effects for Graphics Text: vst_effects()

Another factor affecting text display are special effects, which are set by calling the **vst_effects()** function, as follows:

```
WORD vst_effects( handle, effect )
WORD handle;
WORD effect;
```

The VDI is able to transform text slightly to produce the special effects listed in Table 3.10 and illustrated in Figure 3.23.

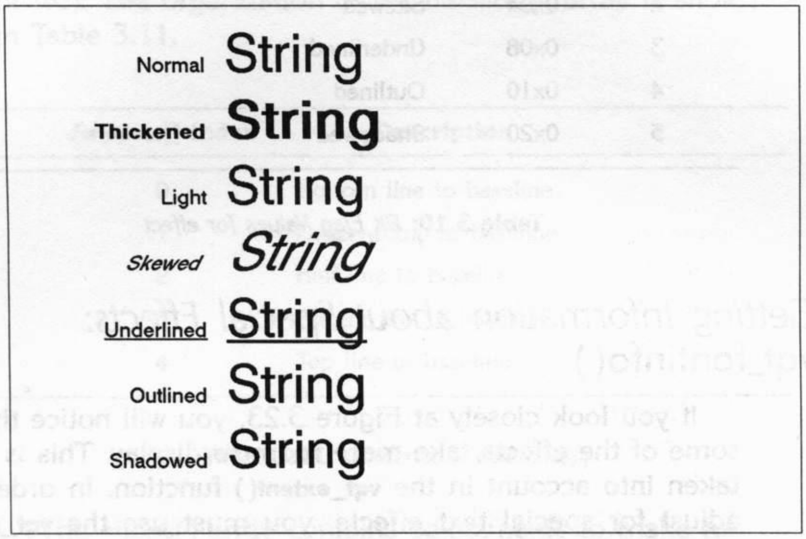


Figure 3.23: *Graphics Text Special Effects*

In order to combine different effects, the value passed into the function in effect is a bit map of requested effects. The effect of bit flag values is documented in Table 3.10. If you want your program to display text in a bold (thickened) and italic (skewed) style, for example, the correct value for effect would be 0x05 (which equals 0x1 OR 0x4). The last two effects (outlining and shadowing) are not available on the current 8086/8088 versions of GEM. Shadowing is not available on the current Atari ST version of GEM. The **vst_effects()** function returns those effects that are actually available.

Bit	Hex Value	Description
0	0x01	Thickened
1	0x02	Lightened Intensity
2	0x04	Skewed
3	0x08	Underlined
4	0x10	Outlined
5	0x20	Shadowed

Table 3.10: *Bit Flag Values for effect*

Getting Information about Special Effects: vqt_fontinfo()

If you look closely at Figure 3.23, you will notice that some of the effects take more room to display. This is not taken into account in the **vqt_extent()** function. In order to adjust for special text effects, you must use the **vqt_fontinfo()** function. The **vqt_fontinfo()** function also returns other useful information, and its prototype looks like this:

```
VOID vqt_fontinfo( handle, pmin_ade, pmax_ade, distances,
                   pmax_width, effects )
WORD handle
WORD *pmin_ade, *pmax_ade, *pmax_width;
WORD distances[5];
WORD effects[3];
```

When the function returns, the variables pointed to by pmin_ade and pmax_ade contain the ASCII decimal equivalent values of the first and last characters available, respectively, for this type face. The variable pointed to by pmax_width contains the maximum cell width, not including special effects.

The distances[ ] array contains offsets of the alignment lines (see Figure 3.21 and Figure 3.22) relative to the baseline of the character cell. All relative distances are positive numbers in the current coordinate system (NDC or RC). The organization of the distances[ ] array is shown in Table 3.11.

distances[] Index	Description
0	Bottom line to baseline
1	Descent line to baseline
2	Half line to baseline
3	Ascent line to baseline
4	Top line to baseline

Table 3.11: *Contents of distances[ ]*

The effects[ ] array contains adjustments to make for the current special effects mode. These values are interpreted as in Table 3.12. Be careful when you use graphics

effects[] Index	Description
0	Current increase of character width due to special effects
1	Left offset
2	Right offset

Table 3.12: *Contents of effects[ ]*

text effects on text that you output to a metafile because the offsets may be very different on different devices. This is why the skewed text displayed by GEM DRAW may appear differently on a screen than it does when the DRAW metafile is output to a printer.

The left and right offsets in the effects[] array are shown in Figure 3.24.

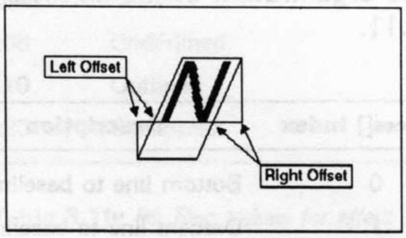


Figure 3.24: *Left and Right Offset for effects[]*

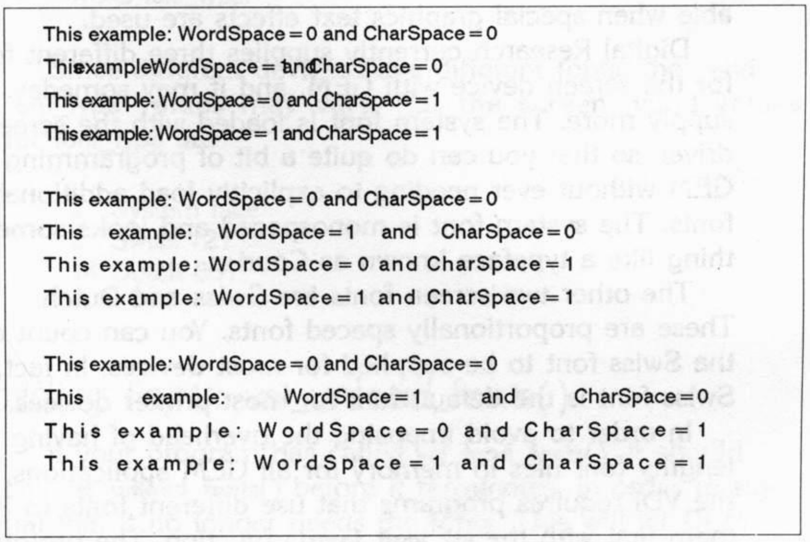
Displaying Justified Graphics Text: v_justified()

To make the job of displaying high-quality graphics text a little bit easier, the VDI provides the function **v_justified**( ). The purpose of this function is to display a text string with extra spacing added or removed between letters and words in the string so that the string's width (also known as *extent*) is of the length specified. The prototype for this procedure is as follows:

```
VOID v_justified( handle, X, Y, string, length, word_space,
                  char_space )
WORD handle;
WORD X, Y;
BYTE *string;
WORD length;
WORD word_space, char_space;
```

The contents of string are printed at position X and Y, as in **v_gtext**( ). However, this function expands or contracts the printed text so that it is as long as the specified length.

v_justified( ) can insert or remove spacing between words or characters. You enable word spacing by setting word_space to a nonzero value. Setting it to 0 tells the function to disable word spacing: that is, to keep the spacing width between words at the normal default value. Similarly, you enable character spacing by setting char_space to a nonzero value, and disable spacing between characters by setting it to 0. If you set both word_space and char_space to 0, this function is the equivalent of **v_gtext**( ). As you examine Figure 3.25, you will notice that if



```
This example: WordSpace = 0 and CharSpace = 0
ThisexampleWordSpace = 1andCharSpace = 0
This example: WordSpace = 0 and CharSpace = 1
This example: WordSpace = 1 and CharSpace = 1

This example: WordSpace = 0 and CharSpace = 0
This    example:    WordSpace = 1    and    CharSpace = 0
This example: WordSpace = 0 and CharSpace = 1
This example: WordSpace = 1 and CharSpace = 1

This example: WordSpace = 0 and CharSpace = 0
This        example:        WordSpace = 1        and        CharSpace = 0
T h i s  e x a m p l e :  W o r d S p a c e = 0  a n d  C h a r S p a c e = 1
T h i s  e x a m p l e :  W o r d S p a c e = 1  a n d  C h a r S p a c e = 1
```

Figure 3.25: *Illustration of v_justified( )*

v_justified( ) has to remove too much space between characters, the characters may overlap each other.

The **v_justified**( ) procedure is actually a generalized drawing primitive (GDP). We discuss GDPs in greater depth in the section entitled "Drawing Shapes—Generalized Drawing Primitives" later in this chapter.

Loading Fonts: vst_load_fonts( )

GEM allows your program to display graphics text in a variety of different type faces or fonts. Different fonts are supplied with different devices, although at least one type face (the Swiss font) should be available on all devices. Unfortunately, because of device characteristics (especially aspect ratio), the same font may have a slightly different appearance on different devices. This will be most noticeable when special graphics text effects are used.

Digital Research currently supplies three different fonts for the screen device with GEM, and it may someday supply more. The system font is loaded with the screen driver, so that you can do quite a bit of programming in GEM without ever needing to explicitly load additional fonts. The system font is monospaced and looks something like a typeface known as Courier.

The other two screen fonts are Swiss and Dutch. These are proportionally spaced fonts. You can count on the Swiss font to be supplied for most devices. In fact, the Swiss font is the default font for most printer devices.

In order to avoid imposing the overhead of having the lengthy font files in memory for all GEM applications, the VDI requires programs that use different fonts to load them first with the **vst_load_fonts**( ) function. The prototype for this function looks like this:

```
WORD vst_load_fonts( handle, select )
WORD handle;
WORD select;
```

This function allows you to load all of the fonts associated with the device pointed to by handle. The function returns the number of additional fonts made available by this call. The select parameter is reserved for future use and should be set to 0.

Setting the Text Face: vst_font()

Once your program has loaded the fonts, it may select between them by using the **vst_font**() call. This function's prototype is as follows:

```
WORD vst_font( handle, font_type )
WORD handle;
WORD font_type;
```

Since different devices have different fonts, the valid font_type values may vary. For the screen, valid values for font_type are

1	System face
2	Swiss 721
14	Dutch 801

Unloading Fonts: vst_unload_fonts()

If your program has called **vst_load_fonts**(), it should call **vst_unload_fonts**() before it terminates, in order to signal that it no longer needs the fonts. This will let GEM free up the memory used by the fonts if no other process is using the fonts. The procedure prototype is as follows:

```
VOID vst_unload_fonts( handle, select )
WORD handle;
WORD select;
```

The select parameter should be equal to the value used for the **vst_load_fonts( )** select parameter (0 for current versions of GEM).

Examples of Text Display

This section contains the programs used to produce some of the figures in this portion of the chapter. We have saved the examples for the end because most of them use several of the functions we have presented.

The first example, shown in Listing 3.11, produced Figure 3.21, "How Text Alignment Works." The program illustrates the use of **vst_alignment( )** as well as **vqt_extent( )**.

```
/* TEXTALGN.C - display text alignment example */

#include "portab.h"

BYTE *x_labels[3] = { " 0:left ", " 1:center ", " 2:right " };
BYTE *y_labels[6] = { " 0:baseline ", " 1:half line ", " 2:ascent line ",
                      " 3:bottom ", " 4:descent ", " 5:top " };

#define LEFT 0
#define CENTER 1
#define HALF 1
#define TOP 5

#define PATTERN 2
#define LIGHT_DITHER_PATTERN 1
#define BLACK 1

#define BIG_LETTERS 36
#define SMALL_LETTERS 10

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
   WORD pxy[4], junk, cur_x, cur_y, incr_x, incr_y, start_x;
   WORD hor_in, ver_in;
   WORD y_lines[6];                      /* table of row positions */

   /* set screen to background dither pattern to highlight text */
   pxy[0] = dx;
   pxy[1] = dy;
   pxy[2] = dx+swidth;
   pxy[3] = dy+sheight;
   vsf_interior( handle, PATTERN );
   vsf_style( handle, LIGHT_DITHER_PATTERN );
   vsf_color( handle, BLACK );
   v_bar( handle, pxy );
```

```
    incr_x = swidth / 3;
    start_x = dx + (2*incr_x) / 3;
    incr_y = sheight / 7;

    /* set up row positions to make text display more pleasing */
    for( cur_y = dy+incr_y, ver_in = 0; ver_in < 6; ++ver_in, cur_y +=incr_y )
        y_lines[ver_in] = cur_y;
    y_lines[1] -= incr_y/3;                /* move these up to make room */
    y_lines[2] -= incr_y/3;
    y_lines[3] += incr_y/2;                /* and move these down */
    y_lines[4] += incr_y/2;

    /* display text in different alignments */
    vst_point( handle, BIG_LETTERS, &junk, &junk, &junk, &junk );
    for( cur_x=dx+start_x, hor_in=0;  hor_in < 3;  ++hor_in, cur_x += incr_x )

        for( ver_in = 0;  ver_in < 6;  ++ver_in )
        {
            vst_alignment( handle, hor_in, ver_in, &junk, &junk );
            v_gtext( handle, cur_x, y_lines[ver_in], "Align");
        }

    /* draw alignment grid and label everything */
    vst_point( handle, SMALL_LETTERS, &junk, &junk, &junk, &junk );

    /* draw and label vertical alignment lines */
    vst_alignment( handle, CENTER, TOP, &junk, &junk );
    for( cur_x=dx+start_x, hor_in=0;  hor_in < 3;  ++hor_in, cur_x += incr_x )
    {
        pxy[0] = pxy[2] = cur_x;
        pxy[1] = dy;
        pxy[3] = dy+sheight;
        v_pline( handle, 2, pxy );
        v_gtext( handle, cur_x, dy, x_labels[hor_in] ); /* label columns */
    }

    /* draw and label horizontal alignment lines */
    vst_alignment( handle, LEFT, HALF, &junk, &junk );
    for( ver_in = 0;  ver_in < 6;  ++ver_in )
    {
        pxy[1] = pxy[3] = y_lines[ver_in];
        pxy[0] = dx;
        pxy[2] = dx+swidth;
        v_pline( handle, 2, pxy );
        v_gtext( handle,dx,y_lines[ver_in],y_labels[ver_in] ); /* label rows*/
    }
}
```

Listing 3.11: *TEXTALGN.C: Display Text Alignment Example*

Our next example, Listing 3.12, loads and selects the Swiss font, and was used to produce Figure 3.22, "Text Alignment Lines for Swiss Font." This example illustrates how **vst_load_fonts( )** and **vst_font( )** are used.

```
/* TEXTSWIS.C - display text alignment for SWISS font lower case letters */

#include "portab.h"

#define SYSTEM_FONT 0
#define SWISS_FONT 2
#define LEFT_ALIGN 0
#define BASELINE 0
#define SMALL_LETTERS 10
#define BIG_LETTERS 72

BYTE *y_labels[6] = { " 0:baseline ", " 1:half ", " 2:ascent ",
                      " 3:bottom ", " 4:descent ", " 5:top " };

WORD print_line( handle, text, cur_x, cur_y, point_size, font_type )
WORD handle, cur_x, cur_y, point_size, font_type;
BYTE *text;
{
    WORD pxy[4], junk, ver_in;
    WORD y_lines[6];                    /* table of row positions */
    WORD distances[5], effects[3];      /* for vqt_font_info() */
    WORD extents[8];                    /* for vqt_extent() */

    vst_font( handle, font_type );      /* use requested font */
    vst_alignment( handle, LEFT_ALIGN, BASELINE, &junk, &junk );
    vst_point( handle, point_size, &junk, &junk, &junk, &junk );
    v_gtext( handle, cur_x, cur_y, text ); /* display the text */

    vqt_font_info( handle, &junk, &junk, distances, &junk, effects );
    y_lines[0] = cur_y;                 /* baseline position */
    y_lines[1] = cur_y - distances[2];  /* halfline position */
    y_lines[2] = cur_y - distances[3];  /* ascent line position */
    y_lines[3] = cur_y + distances[0];  /* bottom line position */
    y_lines[4] = cur_y + distances[1];  /* descent line position */
    y_lines[5] = cur_y - distances[4];  /* top line position */

    vqt_extent( handle, text, extents );
    pxy[0] = cur_x;
    pxy[2] = cur_x + extents[2];        /* right edge of text string */
    vst_font( handle, SYSTEM_FONT );
    vst_point( handle, SMALL_LETTERS, &junk, &junk, &junk, &junk );
    for( ver_in = 5; ver_in >= 0; --ver_in ) {
        pxy[1] = pxy[3] = y_lines[ver_in];
        v_pline( handle, 2, pxy );
        v_gtext( handle, pxy[2], pxy[3], y_labels[ver_in] );
    }
    return distances[0]+distances[4];   /* distance bottom to top lin */
}


VOID draw_rc( handle, dx, dy, screenw, screenh )
WORD handle, dx, dy, screenw, screenh;

{
    WORD cur_x, cur_y;

    vst_load_fonts( handle, 0 );        /* load all available fonts */

    cur_x = dx + (screenw / 32);
    cur_y = dy + (screenh / 4);
    cur_y += print_line( handle, "abcdefghijkl", cur_x, cur_y,
                        BIG_LETTERS, SWISS_FONT );
    cur_y += print_line( handle, "mnopqrst", cur_x, cur_y,
                        BIG_LETTERS, SWISS_FONT );
    print_line( handle, "uvwxyz", cur_x, cur_y, BIG_LETTERS, SWISS_FONT );
}
```

Listing 3.12: *TEXTSWIS.C: Text Alignment for Swiss Font*

Finally, in Listing 3.13 we show the code that used **vst_effects**( ) to produce Figure 3.23, "Graphics Text Special Effects."

```
/* TEXTEFCT.C - illustrate different Graphic Text Special Effects */

#include "portab.h"
#define SYSTEM_FONT 1
#define SWISS_FONT 2
#define LEFT 0
#define RIGHT 2
#define BASELINE 0

BYTE *labels[] = { "Normal", "Thickened", "Light", "Skewed", "Underlined",
                   "Outlined", "Shadowed" };

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD ii, cur_x, cur_y, incr_x, incr_y, junk, effect;

    vst_load_fonts( handle, 0 );         /* load all available fonts */
    incr_x = swidth/40;
    incr_y = sheight/8;
    cur_x = dx + swidth/4;
    cur_y = dy + incr_y;

    for( ii = 0, effect = 1;  ii < 7;  ++ii, cur_y += incr_y ) {
        vst_font( handle, SWISS_FONT );
        vst_point( handle, 36, &junk, &junk, &junk, &junk );
        vst_alignment( handle, LEFT, BASELINE, &junk, &junk );
        if( ii>0 )                       /* no call for normal effect */
        {
            junk = vst_effects( handle, effect );
            effect *= 2;                 /* shift left effects bit by 1 */
            if( effect != junk*2 )
                continue;                /* don't print lines without effect */
        }
        v_gtext( handle, cur_x+incr_x, cur_y, "String" );

        vst_font( handle, SYSTEM_FONT);
        vst_point( handle, 14, &junk, &junk, &junk, &junk );
        vst_alignment( handle, RIGHT, BASELINE, &junk, &junk );
        v_gtext( handle, cur_x, cur_y, labels[ii] );
    }
    vst_effects( handle, 0 );            /* turn off special effects */
    vst_unload_fonts( handle, 0 );       /* clean up fonts */
}
```

Listing 3.13: *TEXTEFCT.C: Graphics Text Special Effects*

RECTANGLES AND FILLED RECTANGLES

The VDI provides a number of functions that display rectangles or affect the way in which rectangular objects are drawn. This section will introduce the most important of these functions.

The functions covered in this section are

vsf_interior()	Set Fill Interior Style
vsf_style()	Set Fill Style Index
vsf_color()	Set Fill Color
vsf_perimeter()	Set Fill Perimeter Visibility
vr_recfl()	Output Rectangle Fill
v_bar()	Output Bar (Rectangle)
v_fillarea()	Output Complex Polygon Fill

Setting the Fill: vsf_interior() *and* vsf_style()

The main new idea in this section has to do with *shading* or *fill.* The VDI allows you to specify how the interior of certain shapes (including rectangles) is drawn. If you want your program to look colorful on a color device and also look readable on a monochrome device, you might be able to use the rectangle fill functions along with color to achieve portability across machines with different color capabilities.

Here is an example of how the fill capability can help your program be portable across color and monochrome devices. Suppose you wanted your program to display a pie chart (like, for instance, the chart shown in Figure 3.31). If the chart is drawn with a solid white interior and black perimeter for each slice, the pie chart would be readable on a color or a monochrome device. If your program were to set the interior to solid nonwhite colors, however, they would be unreadable on monochrome devices because all of the nonwhite colors would be displayed as black and the pie slices would blend together.

You can solve this problem in several ways. The first is to always assume your program will be working with monochrome devices. The disadvantage with this solution is that your program won't be able to use the visually appealing color capabilities of color screens and printers.

The second solution is to check for the number of colors that the screen device supports using the Extended Inquire function **vq_extend()**, discussed later in this chapter. This function returns the number of color planes on the device in work_out[4]. If the device has color, you can print each pie slice in the pie chart in different or alternating colors. If the device doesn't have color or if it's a metafile device, your program can use a solid white interior. Although the solid colors may present the most appealing display on color devices, however, this approach makes your program slightly more complicated. Furthermore, the monochrome metafiles will not display as well if they are displayed on color devices.

A third solution to the problem of portability across color and monochrome devices uses the fill capabilities of the VDI. Instead of drawing the interior of shapes such as pie slices or rectangles in solid colors, the VDI allows your program to specify that the interior is to be filled with a pattern or a type of line hatching. Each pie slice in our example can be filled with different patterns as well as different colors, which would present an effective display on both color and monochrome devices.

The VDI function to set the interior drawing style is called **vsf_interior()**. If the interior fill style you select is the pattern or hatch style, you can choose between a variety of different patterns or hatchings with the **vsf_style()** function. The prototypes for the **vsf_interior()** and **vsf_style()** functions look like this:

```
VOID vsf_interior( handle, fill_style )
WORD handle;
WORD fill_style;
VOID vsf_style( handle, style_index )
WORD handle;
WORD style_index;
```

The interior fill styles you can choose between with the **vsf_interior()** function are listed in Table 3.13.

Value	Meaning
0	Hollow
1	Solid
2	Pattern
3	Hatch
4	Used-Defined style

Table 3.13: *Possible Values for the fill_style Parameter*

Let's discuss each of these patterns briefly. Refer to Figure 3.26 for an illustration of what the different fill styles look like.

In the Hollow fill style, the interior of the rectangle is filled in with white, so that the interior is, in effect, "hollow." This has the same effect as using the Solid fill style with the color set to white. To achieve the effect of only drawing the perimeter of the shape, use Hollow style and Transparent writing mode (set with **vswr_mode()**).

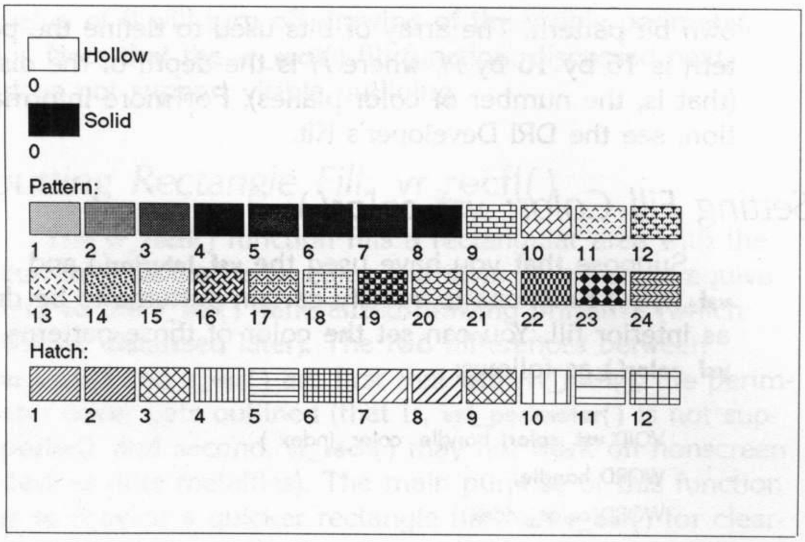


Figure 3.26: *VDI Fill Styles: Hollow, Solid, Pattern, and Hatch*

Next, the Solid fill style sets the interior fill area to the color specified by **vsf_color**( ).

The Pattern and Hatch styles cause different bit patterns to be displayed in the fill area, giving a "shading" effect. The bits are displayed in the fill color (specified with **vsf_color**( )). This kind of shading is particularly appropriate for the kind of problem we just presented—that is, for differentiating regions of the screen for monochrome devices, such as a printer or a black and white screen.

If you use the Pattern or Hatch interior styles, you get to choose the kind of pattern or hatching. The GEM VDI supports a rich variety of 24 different Pattern styles and 12 different Hatch styles (see Figure 3.26). Use the **vsf_style**( ) function to select between these different Pattern and Hatch styles.

Finally, the User-Defined style allows you to create your own bit pattern. The array of bits used to define the pattern is 16 by 16 by *N*, where *N* is the depth of the display (that is, the number of color planes). For more information, see the DRI Developer's Kit.

Setting Fill Color: vsf_color()

Suppose that you have used the **vsf_interior()** and **vsf_style()** functions to specify certain patterns to be drawn as interior fill. You can set the color of those patterns with **vsf_color()** as follows:

```
VOID vsf_color( handle, color_index )
WORD handle;
WORD color_index;
```

The color_index value (see Table 3.4) is used as the color of the filling pixels. If the color_index is out of range, the VDI uses color_index 1 (black).

Setting Fill Perimeter Visibility: vsf_perimeter()

With the **vsf_perimeter()** function, the VDI allows your program to specify whether or not to draw a border around the rectangular shape. The perimeter will be drawn in the fill color set by **vsf_color()**, and it will always be 1 pixel in width. The procedure prototype for **vsf_interior()** is as follows:

```
VOID vsf_perimeter( handle, visible_flag )
WORD handle;
WORD visible_flag;
```

Use this function with visible_flag set to 1 (or any nonzero value) to cause subsequent VDI rectangular fill functions to draw a visible outline around the fill area. Using a value of 0 will turn off drawing of the visible perimeter.

Note that the **vr_recfl()** fill function, discussed next, does not support visible outlining.

Outputting Rectangle Fill: vr_recfl()

The **vr_recfl()** function fills a rectangular area with the currently defined area color and style. It is almost equivalent to the **v_bar()** generalized drawing primitive (which will be discussed later). The two differences between **vr_recfl()** and **v_var()** are first that with **vr_recfl()**, the perimeter never gets outlined (that is, **vsf_perimeter()** is not supported), and second, **vr_recfl()** may not work on nonscreen devices (like metafiles). The main purpose of this function is to provide a quicker rectangle fill than **v_bar()** for clearing rectangular regions such as windows on the screen.

The **vr_recfl()** function can be called like this:

```
VOID vr_recfl( handle, xy_array )
WORD handle;
WORD xy_array[4];
```

Drawing a Bar: v_bar()

v_bar() is the general-purpose function for drawing rectangles. It is a generalized drawing primitive that is available for any device. In this respect, **v_bar()** is unlike **vr_recfl()**, which you should only count on for screen devices. Furthermore, **v_bar()** also has a visible perimeter when so specified with the **vsf_perimeter()** function.

The prototype for this procedure is as follows:

```
VOID v_bar( handle, xy_array )
WORD handle;
WORD xy_array[4];
```

See Listing 3.14 and Figure 3.27 for an example of how **v_bar**() works and what it can do.

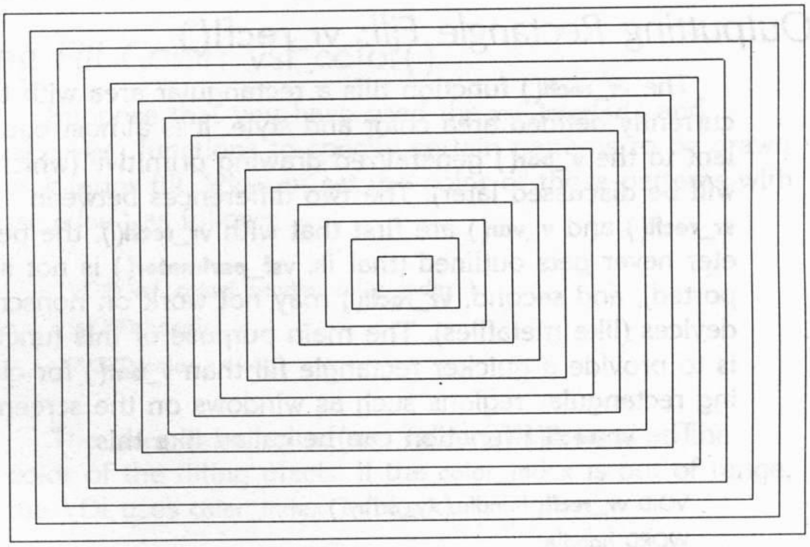


Figure 3.27: *RECTSQAR.C Output: Concentric Squares*

```
/* RECTSQAR.C - draw concentric squares with v_bar() */

#include "portab.h"
#define HOLLOW 0
#define BLACK 1

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD pxy[4], x_incr, y_incr;

    x_incr = swidth/30;
    y_incr = sheight/30;
    pxy[0] = dx;
    pxy[1] = dy;
    pxy[2] = dx+swidth;
    pxy[3] = dy+sheight;
    vsf_interior( handle, HOLLOW );
    vsf_perimeter( handle, TRUE );
    vsf_color( handle, BLACK );
    while( pxy[0] < pxy[2] )
    {
        v_bar(handle, pxy);            /* output hollow square */
        pxy[0] += x_incr;
        pxy[1] += y_incr;
        pxy[2] -= x_incr;
        pxy[3] -= y_incr;
    }
}
```

Listing 3.14: *RECTSQAR.C: Draw Concentric Squares*

Outputting Complex Polygon Fill: v_fillarea()

The next function, **v_fillarea()**, is one of our personal favorites. This function fills the inside part of an arbitrarily complex, possibly self-intersecting polygon with the current fill attributes. The reason it is one of our favorite functions is that the "complex polygon" can look like a child's scribbling, and **v_fillarea()** will choose to fill some of the loops as the "inside" of the figure and leave other loops as the "outside". This function is fun!

The prototype for this function looks like this:

```
VOID v_fillarea( handle, count, xy_array )
WORD handle;
WORD count;
WORD xy_array[2*count];
```

To make certain that the points that have been input in the xy_array contain a polygon, the **v_fillarea()** function causes the last point to be connected with the first. The **v_fillarea()** function does not display a polygon with only one point. If the polygon has no area, it will be displayed as a dot when the fill perimeter visibility is on, and it will not be displayed at all when the perimeter visibility is off.

Listing 3.15 and Figure 3.28 demonstrate the **v_fillarea()** function and its results.

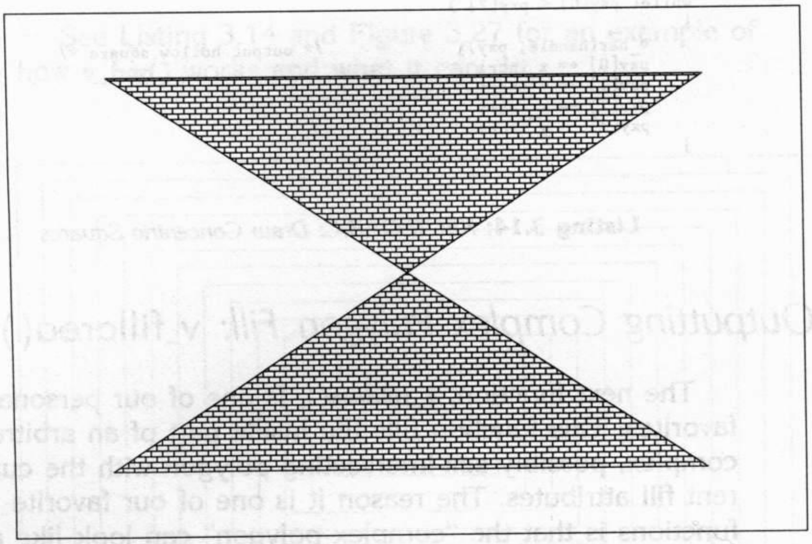


Figure 3.28: *RECTAREA.C Output: Demonstrate v_fillarea( ) Function*

```
/* RECTAREA.C - demonstrate v_fillarea() function */

#include "portab.h"
#define PATTERN 2
#define BLACK 1

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD pxy[8];

    pxy[0] = dx + swidth / 8;              /* specify "hourglass" polygon */
    pxy[1] = dy + sheight / 8;
    pxy[2] = dx + 7 * (swidth / 8);
    pxy[3] = dy + 7 * (sheight / 8);
    pxy[4] = dx + swidth / 8;
    pxy[5] = dy + 7 * (sheight / 8);
    pxy[6] = dx + 7 * (swidth / 8);
    pxy[7] = dy + sheight / 8;
    vsf_interior( handle, PATTERN );
    vsf_style( handle, 9 );
    vsf_color( handle, BLACK );
    v_fillarea( handle, 4, pxy );
}
```

Listing 3.15: *RECTAREA.C: Demonstrate v_fillarea() Function*

DRAWING SHAPES: GENERALIZED DRAWING PRIMITIVES

The VDI provides a number of higher level functions that draw shapes and justified text. These functions handle a considerable amount of detail (including aspect ratio), and they make certain high-level shapes and graphics effects easy to produce. GDPs work with text, line, or rectangle fill attributes, depending on the individual GDP. These functions allow you to specify relatively complicated images with less effort.

We have already introduced two of these functions—**v_justified( )** and **v_bar( )**—in previous sections. The generalized drawing primitives covered in this section are

v_arc( )	Output an Arc
v_pieslice( )	Output a Pie Slice
v_circle( )	Output a Circle
v_ellarc( )	Output an Elliptical Arc
v_ellpie( )	Output an Elliptical Pie Slice
v_ellipse( )	Output an Ellipse
v_rbox( )	Output a Rounded Rectangle
v_rfbox( )	Output a Filled Rounded Rectangle

Drawing an Arc: v_arc( )

The **v_arc( )** function displays a circular arc, corrected for aspect ratio, and is called as follows:

```
VOID v_arc( handle, X, Y, radius, beg_ang, end_ang )
WORD handle;
WORD X, Y;
WORD radius;
WORD beg_ang, end_ang;
```

The beg_ang and end_ang values specify the starting and ending angles of the arc, respectively, in tenths of degrees, where 0 degrees is at 3 o'clock and 90 degrees is at 12 o'clock (see Figure 3.29). Note that **v_arc()** correctly handles aspect ratio to produce a circular arc. The radius is in X units, which may be different than the Y-unit radius (refer to the discussion of aspect ratio at the beginning of this chapter). The arc itself is drawn with line attributes.

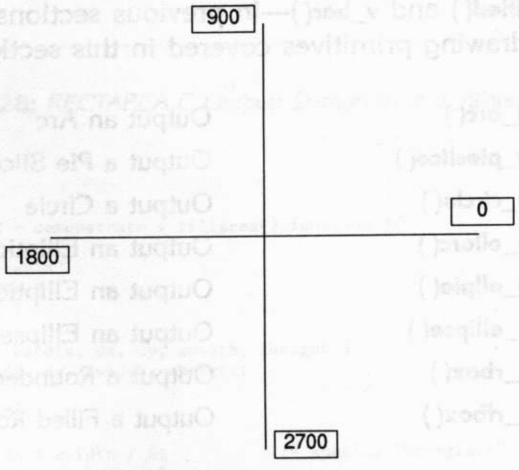


Figure 3.29: *Specification of Angles in the VDI*

Drawing a Pie Slice: v_pieslice()

v_pieslice() is similar to **v_arc()**, except that it uses the rectangle fill attributes to fill in the area between the center and the edge of the specified arc. The function's prototype is as follows:

```
VOID v_pieslice( handle, X, Y, radius, beg_ang, end_ang )
WORD handle;
WORD X, Y;
WORD radius;
WORD beg_ang, end_ang;
```

Drawing a Circle: v_circle( )

This function draws a circle at position X,Y in the current coordinate system (RC or NDC), corrected for aspect ratio, with a radius in X-coordinate units. It can be called as follows:

```
VOID v_circle( handle, X, Y, radius )
WORD handle;
WORD X, Y;
WORD radius;
```

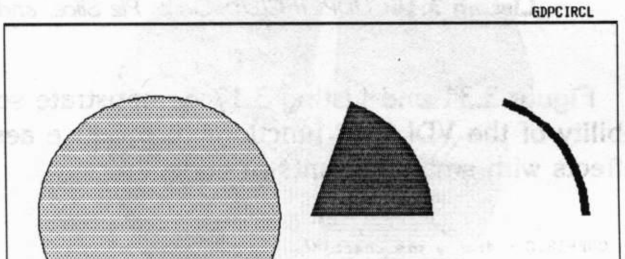


Figure 3.30: *GDPCIRCL.C Output: Circle, Pie Slice, and Arc*

Listing 3.16 and Figure 3.30 show examples of what the last three functions can produce. Notice that the arc has been drawn with a thicker line to emphasize that it uses line attributes. Also notice that the pie slice and circle are using rectangle fill attributes.

```
/* GDPCIRCL.C - draw circle, pieslice, and arc */

#include "portab.h"
#define PATTERN 2
#define BLACK 1

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    /* v_circle() and v_pieslice() uses FILL Attributes */
    vsf_interior( handle, PATTERN );
    vsf_color( handle, BLACK );
    vsf_perimeter( handle, TRUE );
    vsf_style( handle, 1 );
    v_circle( handle, dx+swidth/4, dy+sheight/2, swidth/5 );
    vsf_style( handle, 4 );
    v_pieslice( handle, dx+swidth/2, dy+sheight/2, swidth/5, 0, 700 );

    /* v_arc() uses LINE Attributes */
    vsl_width( handle, swidth/60 );      /* thick line for effect */
    vsl_color( handle, BLACK );
    v_arc( handle, dx+3*(swidth/4), dy+sheight/2, swidth/5, 0, 700 );

    vsl_width( handle, 1 );              /* reset line thickness */
}
```

Listing 3.16: *GDPCIRCL.C: Circle, Pie Slice, and Arc*

Figure 3.31 and Listing 3.17 demonstrate some of the ability of the VDI GDP functions to produce aesthetic effects with small amounts of code.

```
/* GDPPIE.C - draw a pie chart */

#include "portab.h"
#define HATCH 3
#define BLACK 1

#define NSLICES 7
WORD slice_angles[NSLICES] = { 0, 850, 1600, 2250, 2800, 3250, 3500 };

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD ii;
    WORD x_center, y_center, x_incr, y_incr, slice_size, beg_angle, end_angle;

    x_center = dx + swidth/2;
    y_center = dy + sheight/2;
    x_incr = swidth/30;
    y_incr = sheight/18;
    slice_size = swidth / 4;
    vsf_color( handle, BLACK );
    vsf_interior( handle, HATCH );
                                        /* first slice draw apart */
    vsf_style( handle, 1 );
    v_pieslice( handle, x_center + x_incr, y_center - y_incr, slice_size,
        slice_angles[0], slice_angles[1] );

    for( ii=1; ii<NSLICES; ++ii )       /* the rest of the slice_angles */
    {
        vsf_style( handle, ii+1 );      /* different interior pattern */
        beg_angle = slice_angles[ii];
        if( ii < NSLICES-1 )
             end_angle = slice_angles[ii+1];
        else end_angle = slice_angles[0];

        v_pieslice( handle, x_center, y_center, slice_size,
                beg_angle, end_angle );
    }
}
```

Listing 3.17: *GDPPIE.C: Draw a Pie Chart*

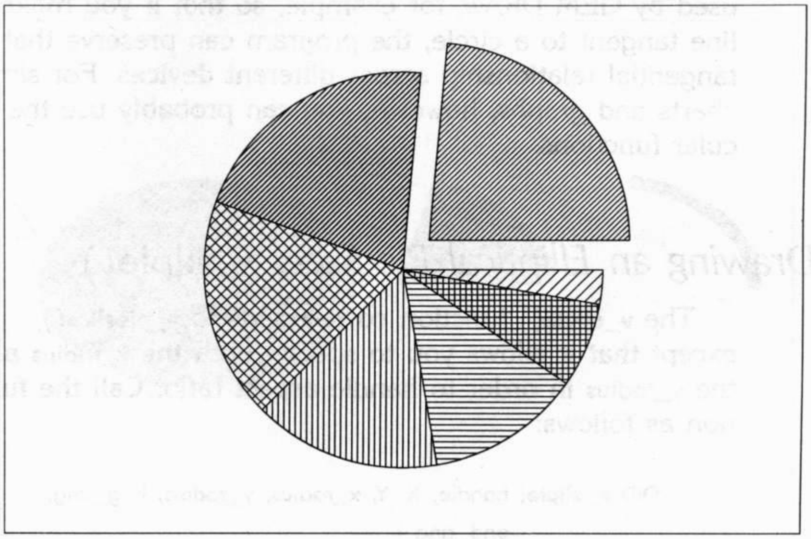


Figure 3.31: *GDPPIE.C Output: Draw a Pie Chart*

Drawing an Elliptical Arc: v_ellarc()

This function corresponds roughly to **v_arc()** except that the arc is not corrected for aspect ratio. Thus, you must specify both the x_radius and the y_radius. The prototype for this function is:

```
VOID v_ellarc( handle, X, Y, x_radius, y_radius, beg_ang,
               end_ang )
WORD handle;
WORD X, Y;
WORD x_radius, y_radius;
WORD beg_ang, end_ang;
```

When it is important for your application to draw different shapes next to each other, you may want to use the elliptical functions to draw circular shapes so that you can adjust all of your shapes for aspect ratio. This technique is used by GEM DRAW, for example, so that if you make a line tangent to a circle, the program can preserve that tangential relationship across different devices. For simple charts and graphs, however, you can probably use the circular functions.

Drawing an Elliptical Pie Slice: v_ellpie()

The **v_ellpie()** function corresponds to **v_pieslice()**, except that it allows you to specify both the x_radius and the y_radius in order to handle aspect ratio. Call the function as follows:

```
VOID v_ellpie( handle, X, Y, x_radius, y_radius, beg_ang,
               end_ang )
WORD handle;
WORD X, Y;
WORD x_radius, y_radius;
WORD beg_ang, end_ang;
```

Drawing an Ellipse: v_ellipse( )

We wrap up our presentation of the circular/elliptical functions with **v_ellipse**( ). Call this procedure as follows:

```
VOID v_ellipse( handle, X, Y, x_radius, y_radius )
WORD handle;
WORD X, Y;
WORD x_radius, y_radius;
```

Figure 3.32 and Listing 3.18 demonstrate the work of some of the circular/elliptical generalized drawing primitives we have just discussed.

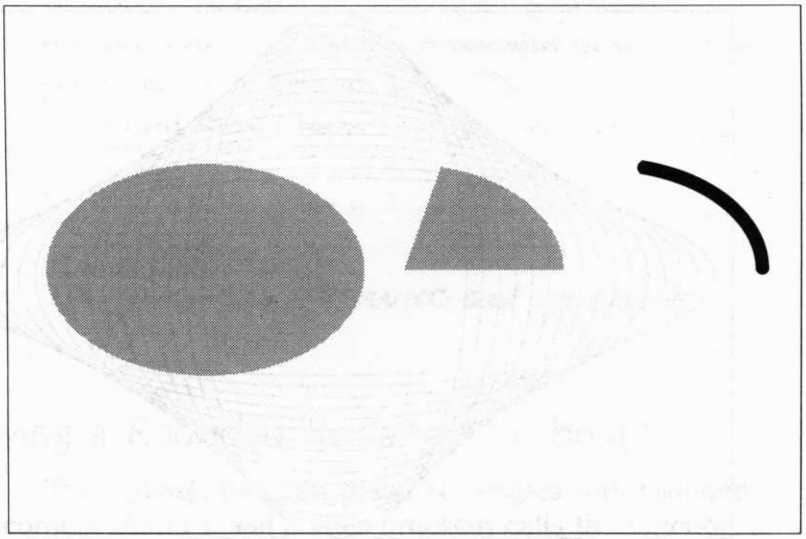


Figure 3.32: *GDPELIPS.C Output: Ellipse, Elliptical Pie Slice, and Arc*

```
/* GDPELIPS.C - draw ellipse, elliptical pieslice, and elliptical arc */

#include "portab.h"
#define PATTERN 2
#define BLACK 1

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    /* v_ellipse() and v_ellpie() uses FILL Attributes */
    vsf_interior( handle, PATTERN );
    vsf_style( handle, 1 );
    vsf_color( handle, BLACK );
    vsf_perimeter( handle, TRUE );
    v_ellipse( handle, dx+swidth/4, dy+sheight/2, swidth/5, sheight/5 );
    v_ellpie( handle, dx+swidth/2, dy+sheight/2, swidth/5, sheight/5, 0, 700);

    /* v_ellarc() uses LINE Attributes */
    vsl_width( handle, swidth/60 );
    vsl_color( handle, BLACK );
    v_ellarc( handle, dx+(3*swidth)/4, dy+sheight/2, swidth/5, sheight/5, 0,
              700 );

    vsl_width( handle, 1 );                 /* reset line width */
}
```

Listing 3.18: *GDPELIPS.C: Draw Elliptical Figures*

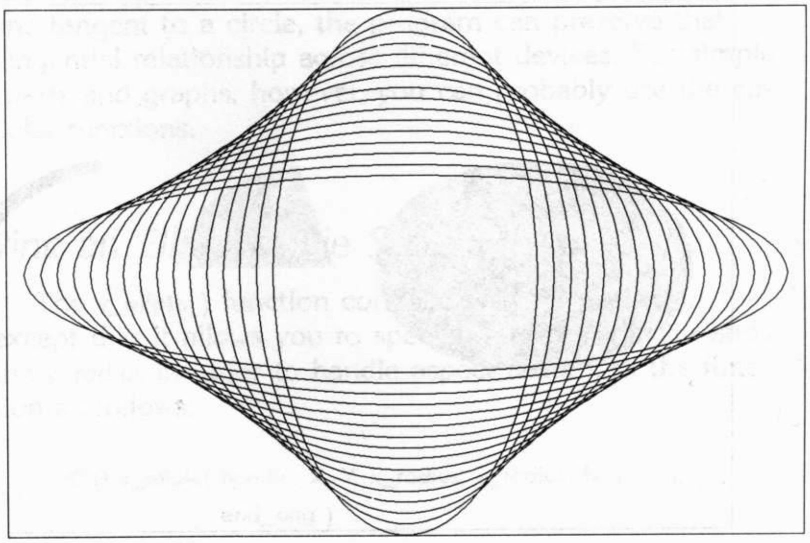


Figure 3.33: *GDPEMANY.C Output: Lots of Ellipses*

The next listing, Listing 3.19, presents some code that demonstrates the creation of different elliptical shapes. Figure 3.33 shows the output.

```
/* GDPEMANY.C - draw many ellipses */

#include "portab.h"
#define NUM_ELIPS 16
#define FUDGE 3
#define HOLLOW 0
#define BLACK 1
#define MD_REPLACE 1
#define MD_TRANSPARENT 2

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    WORD ii, x_center, y_center, x_incr, y_incr, x_size, y_size;

    x_center = dx + swidth/2;
    y_center = dy + sheight/2;
    x_incr = swidth / (3 * NUM_ELIPS);
    y_incr = sheight / (3 * NUM_ELIPS);
    x_size = swidth / 6 - FUDGE;          /* leave a bit of room */
    y_size = sheight / 2 - FUDGE;
    vsf_interior( handle, HOLLOW );
    vsf_perimeter( handle, TRUE );
    vsf_color( handle, BLACK );

    vswr_mode( handle, MD_TRANSPARENT ); /* TRANSPARENT for hollow ellipses */

    for( ii = 0; ii < NUM_ELIPS; ++ii )
    {
        v_ellipse( handle, x_center, y_center, x_size, y_size );
        x_size += x_incr;
        y_size -= y_incr;
    }
    vswr_mode( handle, MD_REPLACE ); /* reset to default */
}
```

Listing 3.19: *GDPEMANY.C: Draw Lots of Ellipses*

Drawing a Rounded Rectangle: v_rbox()

The **v_rbox()** function draws rectangles with rounded corners. As in **v_bar()**, your program calls the function with diagonally opposite corner points.

Here is the procedure's prototype:

```
VOID v_rbox( handle, xy_array )
WORD handle;
WORD xy_array[4];
```

Unfortunately, you cannot specify the radius of the rounded corners, which means that if your rectangle is too small, it won't look very much like a rectangle except on higher resolution devices like printers. This function uses the line attributes, and thus allows you to draw rounded rectangles out of dashed lines. We use different line types in Listing 3.20 and the resulting Figure 3.34 to demonstrate how **v_rbox()** uses line attributes.

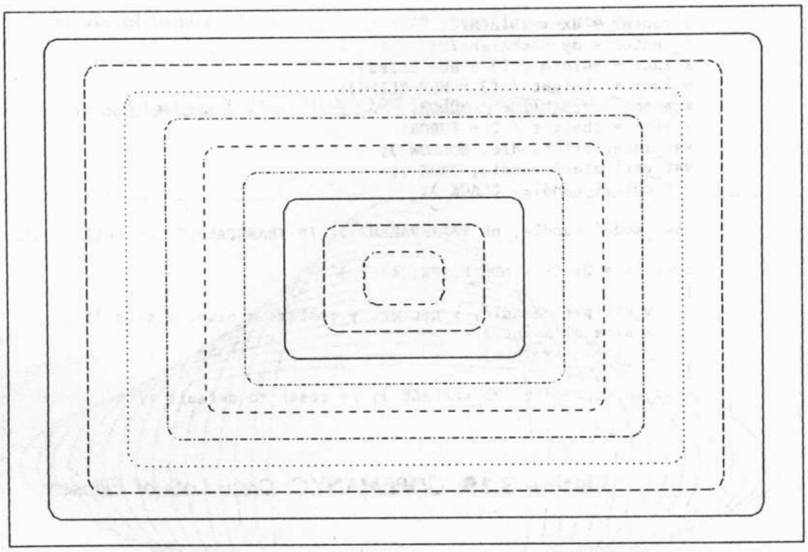


Figure 3.34: *GDPRBOX.C Output: Concentric Rounded Rectangles*

```
/* GDPRBOX.C - draw concentric rounded rectangles with v_rbox() */

#include "portab.h"
#define LINE_TYPE_MAX 6

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
```

```
{
    WORD pxy[4], x_incr, y_incr, line_type;

    line_type = 1;
    x_incr = swidth / 20;
    y_incr = sheight / 20;
    pxy[0] = dx;
    pxy[1] = dy;
    pxy[2] = dx+swidth;
    pxy[3] = dy+sheight;
    while( pxy[0] < pxy[2] )
    {
        pxy[0] += x_incr;
        pxy[1] += y_incr;
        pxy[2] -= x_incr;
        pxy[3] -= y_incr;

        vsl_type( handle, line_type );  /* use different linetypes */
        v_rbox(handle, pxy);            /* output rounded rectangle */

        if( ++line_type > LINE_TYPE_MAX )
            line_type = 1;
    }
    vsl_type( handle, 1 );              /* reset to default */
}
```

Listing 3.20: *GDPRBOX.C: Concentric Rounded Rectangles*

Drawing a Filled and Rounded Rectangle: v_rfbox( )

Use the **v_rfbox( )** function to draw a rounded rectangle using the rectangle fill attributes. See Figure 3.35 for an example of this function's use. Note that if you want to draw a rounded, filled rectangle with a border thicker than one pixel, your program could first call **v_rfbox( )** to draw the inside in the desired fashion, and then call **v_rbox( )** after having set a thicker line width. Note also that we have not included a listing with Figure 3.35 because the program differs only in two lines from Listing 3.20.

The prototype for **v_rfbox( )** is:

```
VOID v_rfbox( handle, xy_array )
WORD handle;
WORD xy_array[4];
```

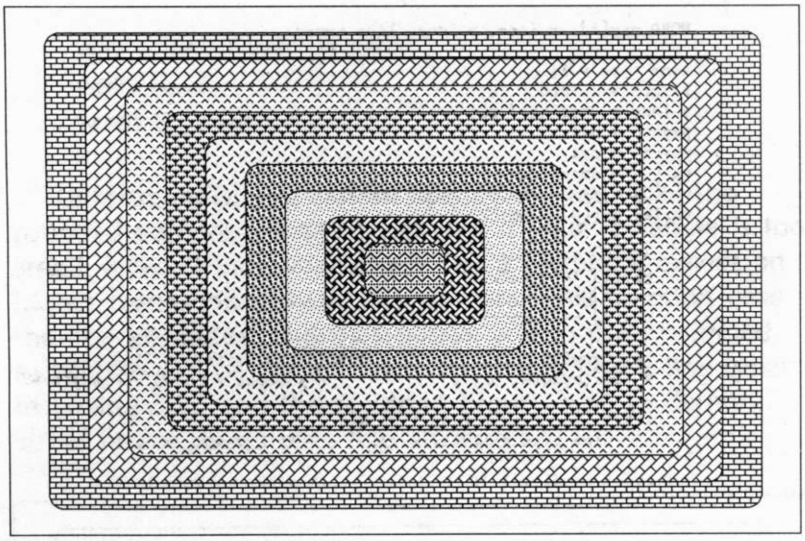


Figure 3.35: *Rounded Filled Rectangles*

RASTER OPERATIONS

Raster operations work on groups of rasters or pixels and are used for icon display, window scrolling and placement, movable cursors, and complex image manipulations. You may know of raster operations by the term *BITBLT,* which we mentioned earlier and which stands for *BIT BLock Transfer.*

The VDI raster operations are used to move blocks of pixels from one part of the screen to another. These operations can also move bit images between the screen and your program's data area, so that your program can efficiently perform direct manipulation of the images.

These raster operations can be very complicated because they offer a lot of function and because they try to be as device-independent as possible. The trouble is that there are so many different ways that pixels are displayed and operated on that it is very inefficient to hide very much detail. For example, **vro_cpyfm()** requires you to know how many color planes exist on your display device if you want to move bit blocks onto or off the screen. To make it easier to write GEM programs, we might prefer to have the memory allocated automatically when we move screen images into our data area. This, however, would impose a level of memory management that the GEM designers felt was not acceptable for all GEM programs, and therefore all of the memory management details are left up to you.

The solution that the GEM designers used to deal with this inherent device diversity was to abstract a certain level of detail and provide the device-specific information in standard forms. To deal with bit images in a standard way, the VDI has the function **vr_trnfm()**, which allows you to transform the bit blocks from a standard, device-independent format into an efficient device-dependent format, and back again. This means that you can always count on a standard pixel representation for complex image manipulation while still enjoying the efficiency of device-specific format. To deal with device-specific information, the VDI provides an inquire function that gives you the information you need. (See the section called "The Extended Inquire Function" later in this chapter for a discussion of this function.)

How GEM Describes a Bit Block: MFDBs

To use the raster operations, you need a way to describe blocks of bits. GEM uses a data structure called a *memory form definition block* (*MFDB*) to describe a bit block. Most of the VDI raster operations require a source MFDB and a destination MFDB. The MFDB either describes a portion of memory called a *raster area* or specifies the physical device (that is, the screen).

Figure 3.36 contains a C structure definition for the MFDB.

```
struct MFDB {

        LONG mem_ptr;

        WORD form_width_pixels;

        WORD form_height_pixels;

        WORD form_width_words;

        WORD is_standard_format;

        WORD num_planes;

        WORD reserved_1;

        WORD reserved_2;

        WORD reserved_3;

};
```

Figure 3.36: *Memory Form Definition Block*

mem_ptr is a long (32-bit) value that points to the start of the raster area. The LONG type can represent an address anywhere in the addressing space of the 8088 or 68000 architectures. (The structure of the raster area is covered in more detail in the next section, "What the Raster Area Looks Like.") If this value is NULL (0), it means that the VDI handle (passed into the function, along with the MFDB pointer) specifies a physical device, and the rest of the MFDB is ignored. Put another way, if mem_ptr is NULL, your source or destination for the BITBLT is the screen. If mem_ptr contains a non-NULL value, on the other hand, the MFDB describes a raster area that is not on the screen but is somewhere in addressable memory.

The form_width_pixels and form_height_pixels parameters are the width and height of the memory area in pixels. The width must be an integral multiple of 16 (the size of a WORD), to allow for bit string alignment on byte boundaries, and thus more efficient data movement. If you are concerned about being limited to WORD-sized bit blocks, take heart. The BITBLT operations **vro_cpyfm**( ) and **vrt_cpyfm**( ) allow you to specify bit areas (rectangles) in any resolution you choose.

form_width_words equals form_width_pixels divided by the WORD size in bits (16) rounded up. Although this information may appear to be redundant, it has been included because this data structure is also used in several internal GEM functions to represent device-specific information that may require extra padding words for raster area alignment.

is_standard_format should contain a nonzero value if the raster area is in standard format (which we'll describe shortly). This flag is reset (to 0) when you use the **vr_trnfm**( ) function to transform the raster area to device specific format. Unfortunately, the VDI does not check this flag before copying this area to the screen, so if you try to BITBLT the raster area before you transform it to device-dependent format, it usually looks strange (how strange depends on your screen device). The reason that the VDI won't transform the raster area while it copies is that some devices have device-specific formats that are very difficult to convert to standard format during a copy operation (most notably the Atari ST). For performance reasons, these devices require transforming from one raster area into another before the copy can take place. Since the VDI tries to minimize the amount of memory management it needs to perform and since the designers of the VDI didn't want the copy functions to fail unexpectedly on different devices, they have left the transformation up to you.

Finally, the num_planes field is the number of planes of color information. Monochrome raster areas only have one plane, while raster areas that handle color information have more than one.

What the Raster Area Looks Like

Let's assume that your MFDB points to a raster area somewhere in memory. This raster area might contain an icon that you've constructed, or it might contain a piece of the screen that you've copied into the raster area. There are two different formats that this raster area might be in.

The first format is device-dependent, and the ordering and state of the bits depends on how your screen device represents its images. The MFDB flag is_standard_format contains 0.

The second format is a standard format. You can manipulate it to suit the needs of your application. This format consists of a set of bit planes (one for monochrome, several for color). Each bit plane is a number (form_width_words) of words of bits for the first row of pixels in the image, followed by the same number of bit words for the second row of pixels, and so forth until there are form_height_pixels groups of words. In each word, the most significant bit (MSB) corresponds to the leftmost bit for that group of bits. The sequence of words that make up one bit plane is immediately followed by the sequence of words for next bit plane.

Color screens require multiple planes to represent color components of individual pixels. Thus, to get the pixel value of the pixel in the top-left corner of a raster area, we collect the first bit in the first plane, and the first bit in the second plane, and the first bit in the third plane, and so forth.

The size of the raster area (in bits) for either device-specific or standard formats equals

form_width_pixels * form_height_pixels * num_planes

For a picture of the standard raster area format, see Figure 3.37. The relationship of pixel values to color values can be found in Table 3.14 for three-plane devices, and Table 3.15 for four-plane devices.

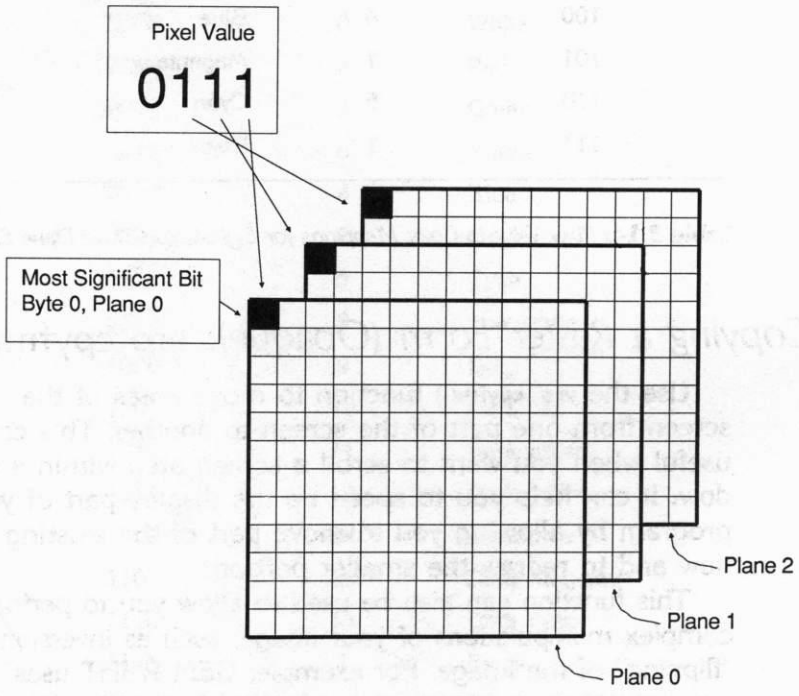


Figure 3.37: *Standard Raster Area Format*

We will cover the following raster operations in this section:

vro_cpyfm( )	Copy Raster, Opaque
vrt_cpyfm( )	Copy Raster, Transparent
vr_trnfm( )	Transform Form

Pixel Value	Color Index	Color
000	0	White
001	2	Red
010	3	Green
011	6	Yellow
100	4	Blue
101	7	Magenta
110	5	Cyan
111	1	Black

Table 3.14: *Pixel Value-to-Color Mappings for Eight-Color, Three-Plane Screens*

Copying a Raster Form (Opaque): vro_cpyfm( )

Use the **vro_cpyfm( )** function to move areas of the screen from one part of the screen to another. This can be useful when you want to scroll a screen area within a window. It can help you to speed up the display part of your program by allowing you to move part of the existing window and to redraw the smaller portion.

This function can also be used to allow you to perform complex manipulations of your image, such as inversion or "flipping" of the image. For example, GEM PAINT uses **vro_cpyfm( )** to copy large parts of your picture between the screen device and memory. It then transforms the image into standard format with the **vr_trnfm( )** function, and it moves bits around to accomplish the manipulation. After the manipulation, PAINT calls **vr_trnfm( )** on the image and then copies it back onto the screen device with **vro_cpyfm( )**.

Call the Opaque Copy Raster Form function as follows:

```
VOID vro_cpyfm( handle, copy_logic, xy_array, psource_mfdb,
                pdest_mfdb )
```

```
WORD handle;
WORD copy_logic;
WORD xy_array[8];
MFDB *psource_mfdb, *pdest_mfdb;
```

Pixel Value	Color Index	Color
0000	0	White
0001	2	Red
0010	3	Green
0011	6	Yellow
0100	4	Blue
0101	7	Magenta
0110	5	Cyan
0111	8	Low white
1000	9	Grey
1001	10	Dark red
1010	11	Dark green
1011	14	Dark yellow
1100	12	Dark blue
1101	15	Dark magenta
1110	13	Dark cyan
1111	1	Black

Table 3.15: *Pixel Value-to-Color Mappings for Sixteen-Color, Four-Plane Screens*

By the way, don't use this function to copy icons and similar monochromatic images onto your screen (the next function we will discuss, **vrt_cpyfm()**, is used for that). The reason for this is that **vro_cpyfm()** does not know how to map a single monochrome plane into a multiple plane (color) raster area. The **vro_cpyfm()** function affects all the planes of color information (via the copy_logic parameter, as we describe next), which means that if you want to copy from the screen, you should make sure you've accounted for all the planes of color when you calculate your memory requirements. Your application can find out how many color planes your screen supports by using the Extended Inquire function, **vq_extend()**. (As explained later in the discussion of the Extended Inquire function, the number of color planes is in work_out[4].) The GEM DEMO program in Chapter 5 contains a good example of using **vro_cpyfm()** and **vq_extend()** in this way.

The copy_logic parameter specifies one of 16 different logical operations that can be performed on the bit planes as they get copied from source to destination. See Table 3.16 for the list of allowable values to this function. Listing 3.21 and Figure 3.38 illustrate the effect of each logic operation obtained by copying the simple "X" figure onto

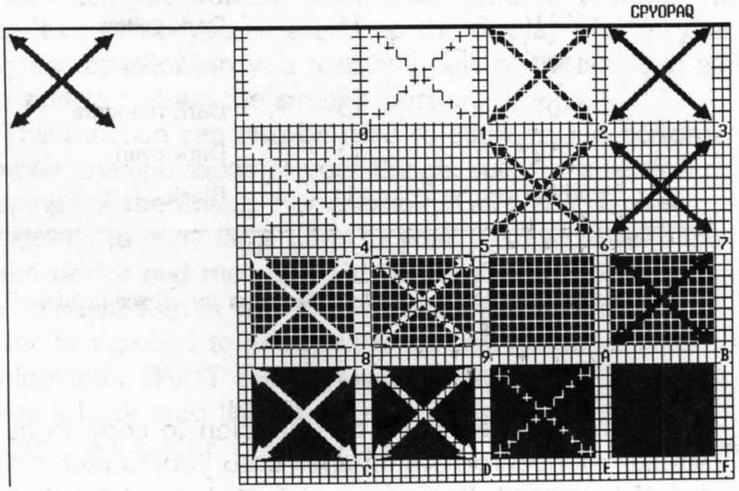


Figure 3.38: *CPYOPAQ.C Output: Demonstrate vro_cpyfm() Operations*

a background grid. You see it here in black and white, but it really gets tricky in color, since the logic operations are handled on a plane-by-plane basis.

```
/* CPYOPAQ.C - illustrates different effects available in vro_cpyfm() */

#include "portab.h"
#include "obdefs.h"

#define RIGHT 2
#define BOTTOM 3
#define SQUARE 0
#define ARROW 1


VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    MFDB s_mfdb;
    WORD pxy[8];
    WORD cur_x, cur_y, x_incr, y_incr, ii, jj, wr_mode;
    BYTE letter[2];

    x_incr = swidth / 6;
    y_incr = sheight / 4;

    pxy[0] = dx + 2*x_incr - 12;          /* prepare backdrop area for exampls*/
    pxy[1] = dy;                          /*   using rightmost 2/3 screen */
    pxy[2] = dx+swidth;
    pxy[3] = dy+sheight;
    vsf_interior( handle, FIS_HATCH );
    vsf_style( handle, 6 );
    vsf_color( handle, BLUE );
    v_bar( handle, pxy );
    letter[0] = '0';                      /* label lower right corner of each */
    letter[1] = 0;                        /*   destination area */
    vst_alignment( handle, RIGHT, BOTTOM, &ii, &ii );
    cur_y = dy + y_incr + 1;
    for( ii=0; ii<4; ++ii )
    {
        cur_x = dx + 3*x_incr;
        for( jj=0; jj<4; ++jj )
        {
            v_gtext( handle, cur_x, cur_y, letter );
            cur_x += x_incr;
            if( *letter == '9' )
                *letter = 'A';
            else (*letter)++;
        }
        cur_y += y_incr;
    }

    vsl_width( handle, 5 );               /* draw big red x in uppr lft corner*/
    vsl_ends( handle, ARROW, ARROW );
    vsl_color( handle, RED );             /* color it RED */
    pxy[0] = dx+3;              pxy[1] = dy+3;
    pxy[2] = dx+x_incr - 12;    pxy[3] = dy+y_incr - 12;
    v_pline( handle, 2, pxy );
    pxy[0] = dx+x_incr - 12;    pxy[1] = dy+3;
    pxy[2] = dx+3;              pxy[3] = dy+y_incr - 12;
    v_pline( handle, 2, pxy );

    cur_y = dy + 1;                     /* copy big X onto backdrop area */
    wr_mode = 0;                        /*  in every writing mode there is */
    s_mfdb.mp = 0L;                     /* Screen MFDB points to phys dev */
    pxy[0] = dx+3;              pxy[1] = dy+3;
    pxy[2] = dx+x_incr - 12;    pxy[3] = dy+y_incr - 12;
    for( ii=0; ii<4; ++ii ) {
        cur_x = dx+2*x_incr;
        for( jj=0; jj<4; ++jj ) {
            pxy[4] = cur_x;             pxy[5] = cur_y;
            pxy[6] = cur_x+x_incr;      pxy[7] = cur_y+y_incr;
            vro_cpyfm( handle, wr_mode, pxy, &s_mfdb, &s_mfdb );
            wr_mode++;
            cur_x += x_incr;
        }
        cur_y += y_incr;
    }
    vsl_ends( handle, SQUARE, SQUARE );
}
```

Listing 3.21: *CPYOPAQ.C: Demonstrate vro_cpyfm( ) Operations*

Table 3.16 illustrates the extensive variety of these functions. Most GEM applications will use only a few of these operations with **vro_cpyfm( )**, and we have described the more useful operations in a terminology consistent with our earlier discussion of writing modes.

You specify the source and destination rectangles within the areas specified by the MFDBs by using the xy_array: the first two vertices (xy_array[0-3]) specify the (X,Y) coordinates of opposite diagonal corners of the source rectangle, and the next two vertices (xy_array[4-7]) specify the destination rectangle. The width of the rectangles does not have to be a multiple of the WORD size (as does the MFDB parameter form_width_pixels).

Don't forget: to make an MFDB point to the screen device, set the mem_ptr field to NULL (0).

Now that we've outlined the major features of this function, what about the fine print? If your rectangles

Mode	Definition	
0	D' = 0	(clears destination)
1	D' = S AND D	
2	D' = S AND [NOT D]	
3	D' = S	(replace mode)
4	D' = [NOT S] AND D	(erase mode)
5	D' = D	
6	D' = S XOR D	(xor mode)
7	D' = S OR D	(transparent mode)
8	D' = NOT [S AND D]	
9	D' = NOT [S XOR D]	
10	D' = NOT D	
11	D' = S OR [NOT D]	
12	D' = NOT S	
13	D' = [NOT S] OR D	(reverse transparent mode)
14	D' = NOT [S AND D]	
15	D' = 1	

S = Source Pixel, D = Destination Pixel, D' = New Destination Pixel

Table 3.16: *copy_logic Values for vro_cpyfm( )*

overlap, the VDI does the right thing: it copies either from the beginning of the rectangles or from the end so that the source image is preserved. If the source and destination rectangles aren't the same size, you will probably be okay on the Intel machines, as the size of the source rectangle is used. On the Atari ST, however, you will find unpredictable results.

Copying a Raster Form (Transparent): vrt_cpyfm()

Compared to **vro_cpyfm()**, the **vrt_cpyfm()** function is really easy. You should use this function only to copy monochrome (single plane) images, such as icons, onto color planes, such as screens. You should use the **vro_cpyfm()** function to move rectangular areas from or around the screen. Call the function as follows:

```
VOID vrt_cpyfm( handle, writing_mode, xy_array, psource_mfdb,
                pdest_mfdb, color_index )
WORD handle;
WORD writing_mode;
WORD xy_array[8];
MFDB *psource_mfdb, *pdest_mfdb;
WORD color_index;
```

This function is different from many of the rest of the VDI functions because you specify the writing mode and pixel color all in one function call. For other groups of VDI operations (for example, displaying polymarkers), the VDI provides separate functions to set these attributes. The **vrt_cpyfm()** function rolls it all into one function.

The writing_mode parameter is just like the Set Writing Mode function, **vswr_mode()**, discussed earlier in this chapter, in that it uses the same four writing modes. The color_index specifies the foreground color, that is, the color that each 1 bit in the source image takes in the destination image (see Table 3.4).

Transforming a Form: vr_trnfm()

Earlier in this discussion of VDI raster operations, we pointed out the differences between device-specific format and standard format for bit images. The **vr_trnfm()** function is used to convert back and forth between these two formats. Most applications that don't do complicated image transformations only use **vr_trnfm()** in the initialization stage to convert icon images from standard to device-specific form.

The procedure prototype for **vr_trnfm()** looks like:

```
VOID vr_trnfm( handle, psource_mfdb, pdest_mfdb )
WORD handle;
MFDB *psource_mfdb, *pdest_mfdb;
```

We need to warn you about doing transforms in-place—that is, calling **vr_trnfm()** with psource_mfdb equal to pdest_mfdb. This technique is safe for small raster areas (like icons), but some machines (most notably the Atari ST) are *very* slow when you transform areas that are significant in size. The reason for this slowness is simply that the color information in these systems is stored in a way that makes it exceedingly difficult to do in-place transforms. You can work around this problem by transforming from one raster area to another, which does, however, require more memory than an in-place transform.

Examples of Raster Operations

Here are a couple of examples demonstrating raster operations. Figure 3.39 displays an expanded graphic representation of the bit image used in Listing 3.22 and Listing 3.23. Listing 3.22 displays this bit image in the middle of the screen, which is shown in Figure 3.40.

Next, Listing 3.23 does some simple animation using **vrt_cpyfm()** and three related versions of the bit image. This program shows the effects of different writing modes, namely replace mode and XOR mode, in Figure 3.41. Two copies of the small bit image start out on the left edge of

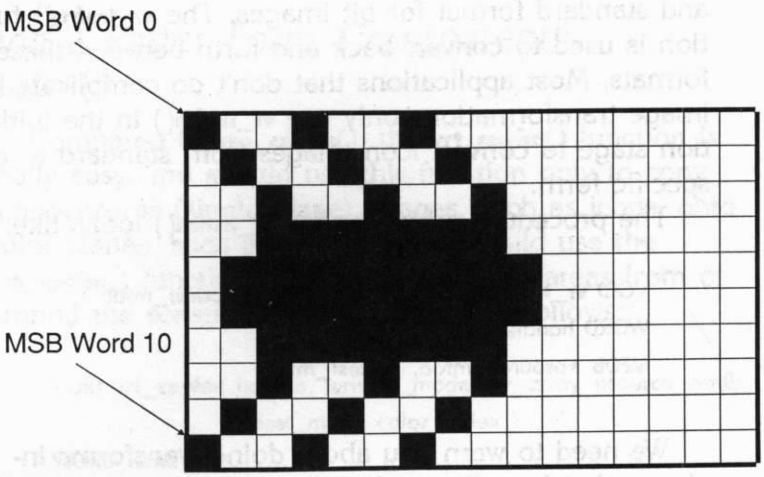


Figure 3.39: *Expanded View of Bit Image*

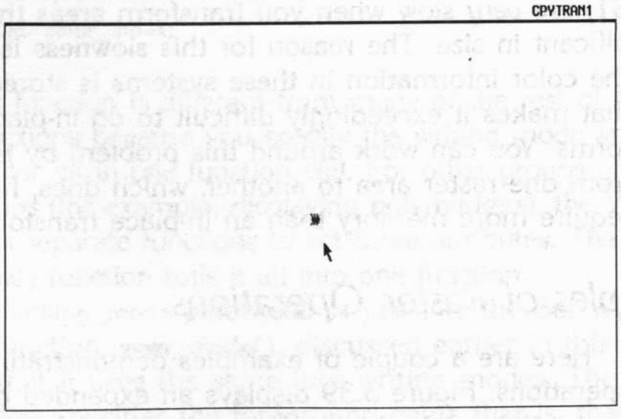


Figure 3.40: *CPYTRAN1.C Output: Display Typical Bug with vrt_cpyfm()*

the screen. The program then copies related images on top of each previous image (for a total of three different images) to give the visual effect of tiny moving legs to the

```
/* CPYTRAN1.C - display a bit image on the screen */

#include "portab.h"
#include "obdefs.h"
#include "machine.h"

#define ImageBytes     2                    /* # bytes in raster area */
#define ImageRows      10                   /* # rows in raster area */
#define ImageBitWidth 11                    /* # significant bits */
WORD image[10] = {      0x9200,0x4900,0x2480,0x3f80,0x7fd0,
                        0x7fd0,0x3f80,0x2480,0x4900,0x9200 };

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    MFDB img_m, scr_m;
    WORD pxy[8], colors[2];

    img_m.mp = ADDR(image);                 /* point to image */
    img_m.fwp = ImageBytes<<3;              /* 32 pixels wide */
    img_m.fh = ImageRows;                   /* 10 pixels high */
    img_m.fww = ImageBytes>>1;              /* 2 words wide */
    img_m.ff = 1;                           /* standard format */
    img_m.np = 1;                           /* only one plane */
    img_m.r1 = img_m.r2 = img_m.r3 = 0;  /* reserved info */

    scr_m.mp = 0L;                          /* Screen MFDB points to phys. dev. */

    colors[0] = 1; colors[1] = 0;           /* Black and White */

    vr_trnfm( handle, &img_m, &img_m ); /* in place transform */

    pxy[0] = 0;                             /* source 'location': only size is */
    pxy[1] = 0;                             /*  important because bit image is */
    pxy[2] = pxy[0] + ImageBitWidth -1; /*  off of the screen */
    pxy[3] = pxy[1] + ImageRows - 1;
    pxy[4] = dx+swidth/2;                   /* destination on screen */
    pxy[5] = dy+sheight/2;
    pxy[6] = pxy[4] + ImageBitWidth;
    pxy[7] = pxy[5] + ImageRows;
    vrt_cpyfm( handle, MD_REPLACE, pxy, &img_m, &scr_m, colors ); /* BITBLT */
}
```

Listing 3.22: *CPYTRAN1.C: Display Bug with vrt_cpyfm()*

image of the bug. The copying location is then moved one pixel to the right and the process is repeated, until the image has "crawled" half way across the screen.

In the XOR mode, each image is copied twice so that the effect of undoing the display is illustrated. As you can see in the figure, the replace mode bug has left a broad track across the carpet, whereas the track of the XOR bug is invisible.

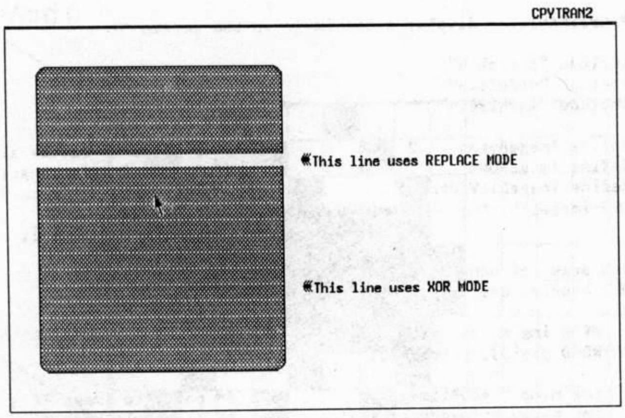


Figure 3.41: *CPYTRAN2.C Output: Demonstrate Replace and XOR Copy*

```
/* CPYTRAN2.C - display two moving bit images on the screen */

#include "portab.h"
#include "obdefs.h"
#include "machine.h"

#define LEFT 0
#define TOP 5

#define ImageBytes      2                   /* # bytes in raster area */
#define ImageRows       10                  /* # rows in raster area */
#define ImageBitWidth 11                    /* # significant bits */

WORD image1[10] = {     0x9200,0x4900,0x2480,0x3f80,0x7fd0,
                        0x7fd0,0x3f80,0x2480,0x4900,0x9200 };
WORD image2[10] = {     0x2480,0x2480,0x2480,0x3f80,0x7fd0,
                        0x7fd0,0x3f80,0x2480,0x2480,0x2480 };
WORD image3[10] = {     0x0920,0x1240,0x2480,0x3f80,0x7fd0,
                        0x7fd0,0x3f80,0x2480,0x1240,0x0920 };

VOID draw_rc( handle, dx, dy, swidth, sheight )
WORD handle, dx, dy, swidth, sheight;
{
    MFDB img_m[3], scr_m;
    WORD ii, cur_x, cur_y, y_incr, pxy[8], colors[2];

    y_incr = sheight / 3;               /* label different writing modes */
    cur_y = dy + y_incr;
    vst_alignment( handle, LEFT, TOP, &ii, &ii );
```

```
        v_gtext( handle, dx+swidth/2, cur_y, "This line uses REPLACE MODE" );
        v_gtext( handle, dx+swidth/2, cur_y+y_incr, "This line uses XOR MODE" );

        pxy[0] = dx+swidth/20;                  /* display backdrop to march thru */
        pxy[1] = dy+sheight/10;
        pxy[2] = dx+(swidth/2)-(swidth/20);
        pxy[3] = dy+sheight-(sheight/10);
        vsf_color( handle, BLACK );
        vsf_interior( handle, FIS_PATTERN );
        vsf_style( handle, 3 );
        v_rfbox( handle, pxy );

                                                /* initialize MFDBs */
        img_m[0].mp = ADDR(image1);             /* point to image1 */
        img_m[1].mp = ADDR(image2);             /* point to image2 */
        img_m[2].mp = ADDR(image3);             /* point to image3 */
        for( ii=0; ii<3; ++ii )
        {
            img_m[ii].fwp = ImageBytes<<3;      /* 32 pixels wide */
            img_m[ii].fh = ImageRows;           /* 10 pixels high */
            img_m[ii].fww = ImageBytes>>1;      /* 2 words wide */
            img_m[ii].ff = 1;                   /* standard fmt (needs transform) */
            img_m[ii].np = 1;                   /* only one plane */
            img_m[ii].r1 = img_m[ii].r2 = img_m[ii].r3 = 0; /* reserved info */

            vr_trnfm( handle, &img_m[ii], &img_m[ii] ); /* in place transform */
        }

        scr_m.mp = 0L;                          /* Screen MFDB points to phys. dev. */

        colors[0] = BLACK; colors[1] = WHITE;

        pxy[0] = 0;                             /* source 'location': only size is */
        pxy[1] = 0;                             /*  important because bit image is */
        pxy[2] = pxy[0] + ImageBitWidth -1;     /*  off of the screen */
        pxy[3] = pxy[1] + ImageRows - 1;

        for( cur_x = dx; cur_x < (dx+swidth/2) - ImageBitWidth; ++cur_x )
        {
            pxy[4] = cur_x;
            pxy[6] = pxy[4] + ImageBitWidth;
            for( ii=0; ii<3; ++ii )
            {
                pxy[5] = cur_y + y_incr;
                pxy[7] = pxy[5] + ImageRows;
                vrt_cpyfm( handle, MD_XOR, pxy, &img_m[ii], &scr_m, colors );
                pxy[5] = cur_y;
                pxy[7] = pxy[5] + ImageRows;
                vrt_cpyfm( handle, MD_REPLACE, pxy, &img_m[ii], &scr_m, colors );
                pxy[5] = cur_y + y_incr;
                pxy[7] = pxy[5] + ImageRows;
                vrt_cpyfm( handle, MD_XOR, pxy, &img_m[ii], &scr_m, colors );
            }
        }
        /* display one last time to leave image on screen */
        vrt_cpyfm( handle, MD_XOR, pxy, &img_m[2], &scr_m, colors );
}
```

Listing 3.23: *CPYTRAN2.C: Demonstrate REPLACE/XOR Copy*

THE EXTENDED INQUIRE FUNCTION

This section contains only one topic: namely, the Extended Inquire function, **vq_extend()**. This function doesn't fit very well with any of the other sections in this chapter, but it was too important to leave out. We have left out many other VDI functions in order to focus on the most useful. The main reason we have included **vq_extend()** is to show you where to find the number of color planes.

The procedure prototype looks like this:

```
VOID vq_extend( handle, info_type, work_out )
WORD handle;
WORD info_type;
WORD work_out[57];
```

Use **vq_extend()** with info_type set to 0 to get the same values that were available when you opened the workstation with **v_opnwk()** or **v_opnvwk()** (see Table 3.3). If you have set info_type to a nonzero value, **vq_extend()** will return the extended device-specific information, shown in Table 3.17. Note that some of these values (for example, work_out[5]) only make sense with VDI functions we have not covered. See the DRI Developer's Kit for more information.

work_out[] Value	**Description**
work_out[0]	Screen Type: 0—Not a screen. 1—Separate alpha and graphic screens. 2—Separate alpha and graphic controllers with common screen.

work_out[] Value	**Description**
	3—Common alpha and graphic controller with separate image memory. 4—Common alpha and graphic controller with common image memory.
work_out[1]	Number background colors in color palette.
work_out[2]	Which text effects are supported.
work_out[3]	0 = scaling not possible, 1 = scaling possible.
work_out[4]	Number of color planes.
work_out[5]	0 = lookup table supported, 1 = not supported.
work_out[6]	Performance factor: number of 16 × 16 pixel raster operations per second.
work_out[7]	Contour fill capability.
work_out[8]	Character rotation capability: 0 = none, 1 = 90 degree increments only, 2 = arbitrary angles.
work_out[9]	Number of writing modes available.
work_out[10]	Highest level of input mode available.
work_out[11]	Text alignment capability: 0 = no, 1 = yes.
work_out[12]	Inking capability flag.
work_out[13]	Rubberbanding capability flag.
work_out[14]	Maximum vertices for polyline, polymarker, or filled area: -1 = no max.

Table 3.17: *Extended Inquire Information (continued)*

work_out[] Value	Description
work_out[15]	Maximum size for intin[]: -1 = no max.
work_out[16]	Number of keys on the mouse.
work_out[17]	Line styles available for wide lines: 0 = no, 1 = yes.
work_out[18]	Writing Modes for wide lines.
work_out[19-57]	Contains zeros.

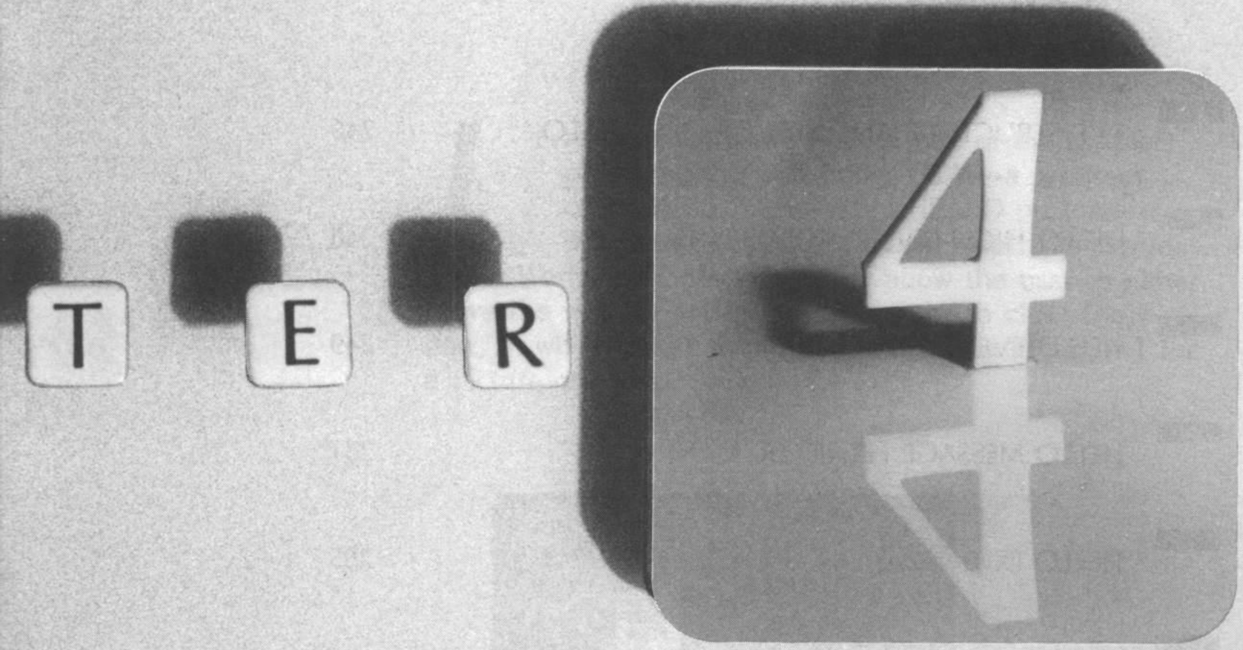
Table 3.17: *Extended Inquire Information*

SUMMING UP

In this chapter we have discussed the VDI functions you can use to produce quality graphic images from your GEM programs. We have grouped together functions by the type of graphical object they deal with: control functions, markers, lines, text, rectangles, generalized drawing primitives, and raster operations. We have provided examples of what these functions produce and the code required to produce them.

In this chapter we have also presented two small driver routines (examprc.c and exampndc.c) that call the small example routines displayed throughout the chapter. We heartily recommend that you use these routines and create your own small VDI examples, or experiment with those we have provided, in order to fully appreciate the power of the VDI. In Chapters 4 and 5, we present specific examples of small GEM applications that use some of these VDI functions (primarily text and raster operations).

C
H
A



GEM SAMPLE PROGRAM: HELLO

In this chapter we look at one of the simpler programs that run under GEM—a program called HELLO. The GEM HELLO program is designed to open a window, display the words "Hello World" in the window, and allow the user to either move the window around the screen or to close the window, in which case the program exits. See Figure 4.1 for a picture of HELLO in action.

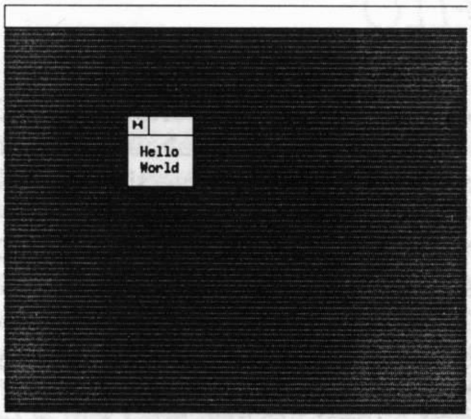


Figure 4.1: *HELLO Application During Execution*

The main purpose of this chapter is to demonstrate the fundamental concepts involved in writing a GEM application that uses windows. We've divided up the HELLO program into several sections, so that we can talk about each section separately. All of these separate pieces are grouped together into a single compiled module using the C preprocessor command, #include. We have used the ".CI" file extension in the name of any C include files containing source code. We have also used certain header files from the GEM Developer's Kit, identified with the ".H" file extension. These files contain a number of macro definitions of constants (also known as *magic numbers*) that GEM uses to specify the behavior of certain functions. We have used all uppercase letters to emphasize that these are constants.

THE STRUCTURE AND SIZE OF GEM HELLO

Before we discuss the specific parts of HELLO, however, we'd like to say a few words about the overall structure and size of the program. Our version of HELLO has been changed slightly from the version provided by DRI in the Developer's Kit. In order to make our HELLO as simple as possible, we have removed some of the complexity of DRI's HELLO—specifically, the conditionally compiled code to make HELLO a desk accessory. Several topics relevant to HELLO are covered in Chapter 6, including the AES and VDI bindings, and the start-up and support routines that are written in assembler.

The general structure of the GEM HELLO application is similar to the structure of the GEM DEMO application in Chapter 5. The HELLO program contains a main entry point that calls an initialization routine, the main application code, and a termination routine. The main application code consists primarily of a single loop that waits for events and delegates these events to the appropriate handling routine. The only events that HELLO waits for are messages from the Screen Manager. Thus, the only event-handling routine in HELLO is a message-handling routine. Finally, GEM HELLO contains code to actually display the message, along with some general-purpose routines.

The GEM HELLO program is based upon a simple program presented in the classic Kernighan and Ritchie text, *The C Programming Language*. Their HELLO.C program has become a standard benchmark used to measure the overhead of a simple printf( ) statement on Unix and DOS C compilers. Here is the entire benchmark HELLO program:

```
main( )
{
    printf("hello, world\n");
}
```

The GEM HELLO program is quite a bit longer than this for several reasons, one of which is that it does more than simply print a line of text. In effect, GEM HELLO must be able to display its message over and over again, so that the user may move the message window around the screen. The benchmark HELLO program only writes its message once to the screen, and then terminates.

Another reason for the size and complexity of GEM HELLO is that GEM was designed to facilitate writing applications that work on many different kinds of machines. This means working with different byte orderings and pointer sizes, as well as working on the smallest machines possible. The original GEM developers felt that the overhead of standard C run-time library support should be eliminated whenever possible. Thus, GEM HELLO doesn't use any of the standard C run-time functions (printf( ), for example).

The novice programmer may be surprised by the size of the executable file of the benchmark HELLO program when it runs on DOS. This is because the printf( ) routine calls in many other run-time library routines to give the programmer as much flexibility as possible. Depending on which C compiler you use, the final executable files for the DOS HELLO program and the GEM HELLO program can be very similar in size.

HELLO HIGH-LEVEL ENTRY POINT

Now that we've presented the overall structure and purpose of the program, we're ready to examine GEM HELLO piece-by-piece. The first piece we'll study is the main high-level entry point routine, called GEMAIN( ), which is shown in Listing 4.1. Standard C compilers use a routine called main( ) to be the entry point of the program. HELLO uses another name because we want to ensure that we are not running with the standard run-time library of our C compiler, along with its associated overhead.

```
/*
**      File:   hmain.ci
**      Purpose: Provides high level entry point for application.
*/

VOID
GEMAIN()
{
        WORD    init_level;                 /* Level of successful init    */

        init_level = hello_init();          /* How successful was init?    */
        if ( init_level == 2 )              /* 2 == completely successful  */
                hello();                    /* Main loop                   */
        hello_term( init_level );           /* Terminate appropriate inits */
}
```

Listing 4.1: *HELLO Entry Point*

The variable init_level, which is returned by hello_init( ), refers to the amount of initialization completed. As we shall see when we examine the hello_init( ) routine, the initialization of a GEM program can fail in several places. For example, the initialization routine may fail when it tries to open a window. Although this is unlikely in the current single-tasking versions of GEM, it may be more likely in future multitasking versions. The technique illustrated here allows the hello_term( ) routine to know how much of the initialization process was performed so that it won't try, for instance, to close a window that was never opened.

If hello_init( ) returns a value of 2, the initialization process succeeded, in which case the main body of code in the hello( ) routine is called.

HELLO EVENT HANDLER

The main loop in the HELLO application is the event handler in the hello( ) routine shown in Listing 4.2. In this routine, the program waits only for a message event and turns the event over to a message handler. This loop continues until the message is decoded to mean that the user wants to terminate the program.

This routine illustrates a couple of important concepts

```
/*
**      File:    hevent.ci
**      Purpose: Application's main loop; handle events.
*/


VOID
hello()
{
        BOOLEAN done;

        done = FALSE;                               /* loop handling user  */
        while( !done )                              /*  input until done   */
        {
                evnt_mesag(ADDR((BYTE *) gl_rmsg));/* wait for message */
                wind_update(BEG_UPDATE);           /* begin window update */
                done = hndl_mesag();               /* handle event message*/
                wind_update(END_UPDATE);           /* end window update    */
        }
}
```

Listing 4.2: *HELLO Event Handler*

covered in Chapter 2. First, note that the program waits (or *blocks*) on the call to the **evnt_mesag**( ) routine, which means that the instructions following the routine are not executed until after the message is received. This blocking is far more efficient than a *polling* method.

Second, remember that when HELLO calls the **wind_update**( ) routine, GEM locks out any other screen update operations until HELLO releases the screen resource. This synchronization method prevents the interruption of graphics operations that may depend on an uninterrupted series of commands to the VDI. When the call signalling the beginning of an *update region* is made, the AES checks to see if any other task has also signalled that it is updating the screen. If the update has already been reserved by another task (for example, by a desk accessory or one of the AES services), the call does not return until the other task completes its use of the screen and signals an end to the update region. When the application's turn has come to update the screen, the AES locks out any other **wind_update**( ) request until the application has signalled that it has completed its VDI operations.

There are a couple of important things to remember about using update regions. First, if the application forgets the **wind_update**(BEG_UPDATE) call, it may seem to work correctly when it is running by itself. If it is working with other GEM tasks such as desk accessories, however, your program may display unpredictable results if another task is also writing to the screen.

Second, if your application returns to the event handler without signalling **wind_update**(END_UPDATE), the AES may behave abnormally and be unable to function properly because it will not be able to write to the screen. One of the symptoms of the update region not being terminated is that you can move the mouse around but none of the window controls seem to work. The reason is that the Screen Manager is blocked, waiting to update the screen.

This can happen when you are debugging your program and the program terminates while in the middle of the update region.

Another important detail about this function is that it illustrates one of the main requirements for writing portable GEM programs. Specifically, most AES functions require that any pointers to strings (or, in this case, to the message return buffer) be LONG values. Thus, the ADDR routine is used to convert a pointer ((BYTE *) gl_rmsg) to a LONG value. Briefly, the reason for this is that GEM AES functions deal with data anywhere in the machine, which requires a 32-bit pointer for the 8086 architecture used by the IBM PC. The whole concept of passing in LONG values into GEM AES bindings is very important and a little tricky. This concept is covered in more detail in Chapter 6.

HELLO MESSAGE HANDLER

Listing 4.3 contains the instructions that decode the received message. Note that HELLO is only interested in a small number of messages sent from the Screen Manager; it ignores any other messages. The Screen Manager has many more messages that it is capable of sending to an application. The kinds of messages sent to an application depend on what window features the application uses. We will describe the window features used by HELLO in a moment. Now we will discuss the messages that HELLO expects to handle.

```
/*
**      File:    hmessag.ci
**      Purpose: Handles all messages sent to application from AES.
*/

```

```
BOOLEAN
hndl_mesag()                          /* Returns TRUE if message indicates */
{                                     /*  user is finished (closes window).*/
        BOOLEAN done;
        WORD    wdw_hndl;

        done = FALSE;
        wdw_hndl = gl_rmsg[3];                /* wdw handle of mesag */
        switch( gl_rmsg[0] )                  /* switch on type msg  */
        {

        case WM_REDRAW:                       /* do redraw wdw contnt*/
                if ( wdw_hndl == appl_whndl )
                        do_redraw ( (GRECT *) &gl_rmsg[4] );
                break;

        case WM_TOPPED:                       /* do window topped    */
                wind_set(wdw_hndl, WF_TOP, 0, 0, 0);
                break;

        case WM_CLOSED:                       /* do window closed    */
                wind_close(appl_whndl);
                done = TRUE;
                break;

        case WM_MOVED:                        /* do window move      */
                wind_set(wdw_hndl, WF_CXYWH, gl_rmsg[4], gl_rmsg[5],
                                        gl_rmsg[6], gl_rmsg[7]);
                wind_get(appl_whndl, WF_WXYWH,
                        &work_area.g_x, &work_area.g_y,
                        &work_area.g_w, &work_area.g_h);
                break;

        default:                         /* ignore any other messages*/
                break;
        } /* switch */
        return(done);
}
```

Listing 4.3: *HELLO Message Handler*

The Window Redraw Message

WM_REDRAW is a message that any GEM application with windows must be prepared to handle. This message is sent to the application whenever the AES discovers that the contents of a window have changed. Since the program must always be able to redisplay its output, you probably need to spend a lot of time carefully designing this part of your program. We discuss the do_redraw( ) routine later in this chapter in the section on "HELLO Display Routines."

While we're talking about the WM_REDRAW message, it might be useful to understand when GEM will ask your program to redraw its display. GEM tries to minimize the redraw requests to your application because it assumes that your program takes a relatively long time to redraw the display. When the user moves the window, GEM tries first to BITBLT (block transfer) the window image from one part of the screen to the new part of the screen. This is possible when the image moves to the right or downward, and when part of the image gets chopped off on the right or bottom edges of the screen. However, when the image moves onto the screen from a position that is off the screen, GEM must send the redraw message to redraw the part of the window that was off the edge.

GEM also has a save buffer that it uses with the menus and alerts. When an alert pops up on the screen, GEM saves the screen area under the alert first. When the alert is finished, this screen area is restored from the save buffer. This allows GEM to avoid sending the redraw message. Due to memory constraints, however, the size of the save buffer is limited to one quarter of the entire screen area. This means that any pull-down menus must be smaller than one quarter of the area of the screen.

When a window is opened with the **wind_open( )** call, or when all or part of a window reappears from underneath another window (which happens when the top window is closed or moved away), GEM sends the redraw message. Sometimes an application sends a redraw message to itself using the AES **appl_write( )** function, in order to update a window that has changed. Since the program must handle redraw requests from GEM, it may be simpler to send a message than to call the application's redraw routine directly.

Window Control Messages

The other two messages, WM_CLOSED and WM_MOVED, are sent when the user interacts with the window control points. HELLO specifies two window control points: a CLOSER and a MOVER.

The CLOSER control point allows the user to click on a part of the window to signal that the window is to be closed. When the user moves the cursor onto the CLOSER control point and presses the button, the Screen Manager *inverts* (turns the white to black, and vice versa) the control point to signal to the user exactly what is being requested. If the user moves the cursor before releasing the button, the Screen Manager returns the control point to its original image and the HELLO application never knows that anything happened. Only when the user presses and releases the button while the cursor rests on the control point will the Screen Manager send the WM_CLOSED message to the application.

For HELLO, closing the window means that the user wants to terminate the program. The **wind_close**( ) routine removes the window and its contents from the screen. Later, in the termination code, the window is deleted. To signal an end to the program, the done flag is set and later returned to the caller of the hndl_mesag( ) routine.

The MOVER control point allows the user to click-drag on the title bar of the window to move the window to another portion of the screen. The user moves the cursor to the title bar of the window, presses the button, and then moves the cursor to where she wants the window and releases the button. The Screen Manager handles the visual feedback of a window outline moving with the cursor. Since the window outline follows the cursor around the screen, the user gets the (intentional) impression that she is dragging the box around the screen (hence the term "click-drag"). The Screen Manager handles all of the visual feedback, and only sends the WM_MOVED message when the user releases the button.

The **wind_set()** call tells GEM that the application wants to move the window. GEM does not move the window until the application has told it to with this call. Once the window has been moved, the **wind_get()** call gets the new work area values. This call is much easier than having the program recalculate the position of the new work area, as GEM knows how big the window's borders are and the application does not.

TWO DIFFERENT KINDS OF WINDOW COORDINATES

One of the tricky aspects about working with windows is distinguishing between the outside coordinates of the window (WF_CXYWH) and the inside coordinates (the working area, WF_WXYWH). The outside coordinates are generally used to position the entire window on the screen. The inside coordinates are used to set clipping rectangles during display.

HELLO INITIALIZATION AND TERMINATION

We've examined events and messages, and we've mentioned that the way in which the windows are created determines what messages the application should be prepared to handle. In this section, we will be looking at the initialization routine itself. Understanding this routine reveals a great deal about how the application meshes with the GEM environment.

Because they are so closely interrelated, the global variables, definitions, initialization, and termination routines are all grouped together in Listing 4.4. The hello_init( ) and hello_term( ) routines work together to initialize and clean up the GEM environment in a coordinated fashion. The coordination is provided via the return value of hello_init( ) and input parameter of hello_term( ) (see the init_level variable in GEMAIN( )).

```
/**********************************************************************/
/*       File:   hello.c                                              */
/**********************************************************************/
/*                                                                    */
/*  The  source code  contained in  this listing is a non-copyrighted */
/*  work which  can be  freely used.  In applications of  this source */
/*  code you  are requested to  acknowledge Digital Research, Inc. as */
/*  the originator of this code.                                      */
/*                                                                    */
/*  Author:     Tom Rolander                                          */
/*  PRODUCT:    GEM Sample Desk Top Accessory                         */
/*  Module:     HELLO                                                 */
/*  Version:    November 15, 1985                                     */
/*  Mods by:    Bill Fitler                                           */
/*                                                                    */
/**********************************************************************/



                                  /****************************************/
                                  /* GEM Toolkit standard include files   */
                                  /****************************************/
#include "portab.h"               /* Portable coding conventions          */
#include "machine.h"              /* Machine dependent conventions        */
#include "obdefs.h"               /* Object definitions                   */
#include "gembind.h"              /* GEM magic numbers                    */
                                  /****************************************/
                                  /* Prototype function declarations      */
                                  /****************************************/
#if 000
#include "tgembind.h"             /* Tiny GEM bindings prototypes         */
#include "tvdibind.h"             /* Tiny VDI bindings prototypes         */
#endif
                                  /****************************************/
                                  /* Appl's definitions & declarations    */
                                  /****************************************/
#include "hdefines.ci"            /* Application specific magic numbers   */
#include "hdecls.ci"              /* Application's global variables       */
                                  /****************************************/
                                  /* Application's main code              */
                                  /****************************************/
#include "hlocal.ci"              /* Application support routines         */
#include "hdisplay.ci"            /* Application display routines         */
#include "hmessag.ci"             /* Application message handling         */
```

```
#include "hevent.ci"            /* Application event handling          */
#include "hinit.ci"             /* Application initialization routine  */
#include "hterm.ci"             /* Application termination routine     */
#include "hmain.ci"             /* Application entry point             */
                                /****************************************/
```

```
/*
**      File:   hdecls.ci
**      Purpose: Provides global variables for application.
*/

WORD    gl_wchar;                       /* character width              */
WORD    gl_hchar;                       /* character height             */
WORD    gl_wbox;                        /* box (cell) width             */
WORD    gl_hbox;                        /* box (cell) height            */
WORD    gem_handle;                     /* GEM vdi handle               */
WORD    vdi_handle;                     /* hello vdi handle             */
GRECT   work_area;                      /* current window work area     */
WORD    gl_rmsg[8];                     /* message buffer               */
WORD    gl_xfull;                       /* full window 'x'              */
WORD    gl_yfull;                       /* full window 'y'              */
WORD    gl_wfull;                       /* full window 'w' width        */
WORD    gl_hfull;                       /* full window 'h' height       */
WORD    appl_whndl = 0;                 /* hello window handle          */
WORD    type_size;                      /* system font cell size        */
#define MESS_NLINES     2               /* maximum lines in message     */
#define MESS_WIDTH      7               /* maximum width of message     */
BYTE    *message[] =                    /* message for window           */
{
        " Hello ",
        " World ",
        0                               /* null pointer terminates inpt*/
};
BYTE    *wdw_title =    "";             /* blank window title           */
```

```
/*
**      File:   hdefines.ci
**      Purpose: Provide meaningful symbols for magic numbers.  Note
**               that 'gembind.h' provides these symbols for most AES
**               specific values.
*/

                                /* 'graf_mouse()' mouse shape equates */
#define ARROW           0
#define HOUR_CLASS      2

                                                /* handle of Desktop */
#define DESK            0

                                        /* 'wind_update()' state values */
#define END_UPDATE      0
#define BEG_UPDATE      1

                                /* 'v_openvwk()' paramater array sizes */
#define V_OVW_IN_SIZE   11
#define V_OVW_OUT_SIZE  57

                            /* application uses these window features */
#define W_FEATURES      (NAME+CLOSER+MOVER)


```

```
/*
**      File:    hinit.ci
**      Purpose: Initialize application and its environment.
*/

WORD
hello_init()                            /* Returns number of successful*/
{                                       /*     init operations: max==2  */
        WORD    ii;
        WORD    work_in[V_OVW_IN_SIZE]; /* Input values for v_opnvwk() */
        WORD    work_out[V_OVW_OUT_SIZE];/* Return val for v_opnvwk()  */
        GRECT   box;

        if( appl_init() == -1 )         /* Initialize AES libraries    */
                return 0;               /* Return init level 0 if fail */

        for (ii=0; ii<10; ii++)         /* 'v_opnvwk()' input parms    */
                work_in[ii] = 1;        /* Use default values          */
        work_in[10] = 2;                /* Use Raster Coordinates      */
        gem_handle = graf_handle(&gl_wchar,&gl_hchar,&gl_wbox,&gl_hbox);
        vdi_handle = gem_handle;
        v_opnvwk(work_in,&vdi_handle,work_out); /* open virtual workstn*/
        type_size = work_out[48];       /* Get system font hbox size   */
        if (vdi_handle == 0)            /* Make sure v_opnvwk() was ok */
                return 0;               /* Return init level 0 if fail */
                                        /* Get the desktop coords      */
        wind_get(DESK, WF_WXYWH,
                &gl_xfull, &gl_yfull, &gl_wfull, &gl_hfull);
                                        /* Set up the appl's window    */
        graf_mouse(HOUR_GLASS, 0L);     /* This may take a while       */
        appl_whndl = wind_create(W_FEATURES,
                        gl_xfull, gl_yfull, gl_wfull, gl_hfull);
        if (appl_whndl == -1)           /* If wind_create() failed     */
        {
          form_alert(1,ADDR(
          "[3][Fatal Error !|Window not available|for Hello.][ Abort ]"
          ));
          return 1;                     /* Return level 1 on failure   */
        }
        wind_set(appl_whndl, WF_NAME,
                LLOWD(ADDR(wdw_title)), LHIWD(ADDR(wdw_title)),
                0, 0);
        wdw_size(&box, MESS_WIDTH, MESS_NLINES);
        wind_open(appl_whndl, box.g_x, box.g_y, box.g_w, box.g_h);
        wind_get(appl_whndl, WF_WXYWH,  &work_area.g_x, &work_area.g_y,
                                        &work_area.g_w, &work_area.g_h);
        disp_mesag(message, &work_area);
        graf_mouse(ARROW,0L);
        return 2;                       /* Return level 2 if all okay  */
}

```

```
/*
**      File:    hterm.c
**      Purpose: Cleanup and terminate the program.
*/

```

```
VOID
hello_term(i_lev)
WORD i_lev;                                     /* How much of init worked      */
{
        switch( i_lev )                         /* NOTE! This switch statement */
        {                                       /*   contains no 'breaks' - fall*/
                                                /*   through is intentional.    */
                                                /********************************/
        case 2:                                 /* Complete initialization      */
                wind_delete(appl_whndl);/* Delete application's window */
                                                /********************************/
        case 1:                                 /* Failed on wind_create()      */
                v_clsvwk( vdi_handle ); /* Close virtual work station  */
                appl_exit();                    /* AES library termination      */
                                                /********************************/
        case 0:                                 /* Failed on v_opnvwk()         */
                ;
        }
}

```

Listing 4.4: *HELLO Global Variables, Initialization, and Termination*

GEM HELLO uses many global variables, as seen in lines 1 through 28 of file HDECLS.CI. The program might have used fewer global variables by passing more parameters into each routine. You must be careful, however, when you pass many parameters or declare many local variables in C, because all local variables are allocated on the stack. In order to minimize the size of your program, you will want to minimize the amount of space allocated to your program's stack in the start-up code. If your program uses more local variables and thus more stack space, you must make sure that there is sufficient room for the stack allocated in your start-up code. We used the PROSTART.A86 module in the Developer's Kit for the start-up code for HELLO.

Some of the #defines could have been in the file GEMBIND.H, specifically the #defines in lines 9 through 21 of the file HDEFINES.CI. Although the GEM developers did not add these definitions, you might want to add them to your copy of GEMBIND.H. The source code was included as a tool for building GEM programs, and we believe that adding appropriate definitions to these files is like sharpening your tools.

Any application that uses the GEM AES calls must use **appl_init**( ) (see line 14 of file HINIT.CI) to initialize internal AES data structures. **appl_init**( ) returns ap_id, which is an integer that uniquely identifies the current instance of program execution. ap_id can be used for *interprocess communication* or to make unique temporary file names, and it should be tested (as shown in this example) in case the **appl_init**( ) function fails. Although the call doesn't fail in current versions of GEM, **appl_init**( ) may fail in future versions of GEM where the user has started up a number of other applications and there are too many for the internal AES structures to accommodate.

Opening the Virtual Workstation

The code in lines 17 through 25 initializes the virtual workstation that the application uses and also inquires about device-dependent information. As explained in Chapter 3, the purpose of the virtual workstation is to simulate complete ownership of the screen device for the application. The Open Virtual Workstation call, **v_opnvwk**( ), is required in order to make any output calls to the VDI.

The input to the **v_opnvwk**( ) routine requires three parameters. The first of these is work_in[ ], an array of parameters that initializes the characteristics of the VDI variables such as line type and color. The default value of 1 is intended to be a "lowest common visible denominator" of workstation attributes. In other words, setting the value to 1 is the convention for "do something predictable," like setting the line type to solid and the color to black. The work_in[10] parameter indicates what coordinate system is to be used for the screen. Raster coordinates are preferable to normalized device coordinates, and they should be used in order to give your program access to efficient raster operations.

The second parameter is used both as an input and an output parameter. The value passed into the routine is gem_handle, which is the physical VDI handle of the screen device returned from **graf_handle**( ) in line 20. The value returned from **v_opnvwk**( ) is the handle of the virtual workstation created by the call. This handle is used, of course, for all of the rest of the VDI calls to identify the appropriate virtual workstation. Thus, an application could conceivably open up several virtual workstations on the screen device in order to, for example, use several different type sizes and/or colors without having to reset the device attributes each time.

The third input parameter is an array of output values that describes the physical device. In particular, work_out[48] contains the maximum height (in pixels, as HELLO specified raster coordinates) of any character in the default system font. This value is used to calculate the size of HELLO's window.

THE TWO KINDS OF INFORMATION IN work_out[ ]

The DRI GEM Developer's Kit documentation has an interesting method of documenting the ***v_opnwk( )*** *and* ***v_opnvwk( )*** *output parameters. Specifically, the* work_out[ ] *array is divided into two arrays:* intout[ ] *and* ptsout[ ]. *The main reason for two arrays was that the* intout[ ] *array contains control/status information, whereas the* ptsout[ ] *array contains coordinate values. For efficiency reasons, the normalized device coordinate values were chosen to fit in WORD (16-bit) values, and most devices supported by GEM have pixel ranges that can also be represented as WORD values. Since both coordinates and control information are WORD values, these arrays were concatenated together, also for efficiency reasons.*

Watch out! Since intout[ ] *stops at element 44,* work_out[48] *corresponds to* ptsout[3] *in the document.*

Building the Window

After opening a virtual workstation, the program gets the DESKTOP screen coordinates (lines 27 and 28) in order to get the size of the screen. A GEM application can assume that there is always one opened window with window handle 0 and a visible arrow cursor. The DESKTOP window is never closed. This window represents the largest window that an application can display on the screen. (Note that the DESKTOP here is not the GEM DESKTOP application but the window with a handle of 0. The DRI documentation also refers to this window as the DESK window.)

In lines 29 through 48, HELLO creates the window and writes the initial message onto the screen. First, it changes the shape of the cursor from an arrow to an hour glass by calling **graf_mouse( )** to indicate that a time-consuming operation will be occurring. This is a GEM convention (as opposed to a requirement) that should be used whenever the application performs an operation that may take a noticeable amount of time. See Chapter 6 for more information on GEM program conventions.

Next, HELLO creates a window with the **wind_create( )** call. Nothing is displayed on the screen, however, nor is a window opened. Instead, **wind_create( )** reserves space in the Screen Manager's data structures and initializes internal variables, as well as specifying the features of the window. For HELLO, these features include a title bar (NAME) and two control points, a CLOSER and MOVER, and the features indicate what is drawn in the borders around the window, as well as which messages can be sent from the Screen Manager to the application. (We talked about these messages in an earlier section called "Window Control Messages.") The return value (appl_whndl) is used whenever the program refers to the window. Your application can open more than one window, if the windows are available—that is, if they are not used up by desk accessories or other applications.

Should the window be unavailable, the application notifies the user with the **form_alert( )** function, which specifies a very rigid format for giving textual feedback to the user.

Figure 4.2 shows what this particular alert looks like. The format for these parameters was discussed in Chapter 2.

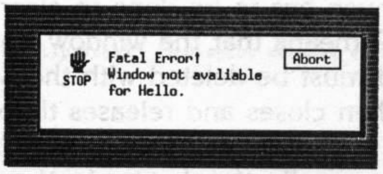

```
Fatal Error!
Window not avaliable
for Hello.
```

Figure 4.2: *form_alert( ) Sample Output*

Once the window has been created, the application uses a **wind_set**( ) call to set the NAME region to a string value. In HELLO, the string wdw_title is empty (that is, it contains a NULL character). Although it seems redundant to give the window a null name, the reason for this is that the MOVER control point requires that the NAME region exist and have some value.

wdw_size( ) is a local routine that calculates the size of the window required to display a message and makes the appropriate window calls. It also sets the initial window position to the center of the DESKTOP window. wdw_size( ) is found in Listing 4.5, which contains the display routines and which we'll discuss in a moment.

Finally, on line 44, the window is opened and displayed for the first time. However, only the window's borders are drawn. HELLO must first find the region actually inside the window's borders with the **wind_get**( ) call, and then draw the inside of the window with the disp_mesag( ) routine, called at line 47. The mouse form is then turned back into an arrow, and the initialization routine signals a successful completion.

HELLO Termination

The hello_term( ) routine in Listing 4.4 cleans up the internal GEM data areas. The i_lev variable specifies how much of the initialization was successfully completed, and thus how much has to be done to clean up. For instance, a value of 2 means that the window was successfully created and must be deleted with the **wind_delete**( ) call. **v_clsvwk**( ) then closes and releases the virtual workstation. Finally, the **appl_exit**( ) call should be used to notify the AES that the application is terminating. This allows the AES to perform certain housekeeping chores, such as deallocating internal data structures.

HELLO DISPLAY ROUTINES

A number of routines are involved with the details of the screen display. These routines, listed in Listing 4.5, include the following:

wdw_size( )	Calculates the necessary window size.
disp_messag( )	Calls VDI to display text strings in window.
do_redraw( )	Calls disp_messag( ) routine as needed when redrawing the screen.

```
/*
**      File:    hdisplay.c
**      Purpose: Routines to display the message
*/

VOID
wdw_size(box, w, h)         /* Compute window size for given w * h chars */
GRECT   *box;
WORD    w, h;
{
        WORD    pw, ph;

        vst_height(vdi_handle, type_size,
                &gl_wchar, &gl_hchar, &gl_wbox, &gl_hbox);
        pw = w * gl_wbox + 1;
        ph = h * gl_hbox + 1;
        wind_calc(WC_BORDER, W_FEATURES,
                gl_wfull/2-pw/2, gl_hfull/2-ph/2, pw, ph,
                &box->g_x, &box->g_y, &box->g_w, &box->g_h);
}



VOID
disp_mesag(strptr, clip_area)  /* Display message applying input clip   */
BYTE    **strptr;
GRECT   *clip_area;
{
        WORD    pxy[4];
        WORD    ycurr;

        set_clip(vdi_handle, TRUE, clip_area); /* Turn clipping on      */
        vsf_interior(vdi_handle, 1);       /* 1==lowest intensity pattern */
        vsf_color(vdi_handle, WHITE);      /* Color for polygon fill    */
        grect_to_array(&work_area, pxy);
        vr_recfl(vdi_handle, pxy);         /* Clear entire message area */

        vsl_color(vdi_handle,BLACK);       /* Color for line drawing    */
        vswr_mode(vdi_handle,MD_REPLACE);/* Use REPLACE writing mode    */
        vsl_type (vdi_handle,FIS_SOLID);/* Use SOLID line type          */
        ycurr = work_area.g_y - 1;         /* Start typing here         */
        while (*strptr)                    /* Loop through text strings */
        {
                ycurr += gl_hbox;
                v_gtext(vdi_handle, work_area.g_x, ycurr, *strptr);
                strptr++;
        }
        set_clip(vdi_handle, FALSE, clip_area); /* Turn clipping off    */
}



VOID
do_redraw ( area )             /* Walk rectangle list and display message */
GRECT   *area;
{
        GRECT   box;

        graf_mouse(M_OFF, 0L);
        wind_get(appl_whndl, WF_FIRSTXYWH,
                &box.g_x, &box.g_y,
                &box.g_w, &box.g_h);
        while ( box.g_w != 0  &&  box.g_h != 0 )
        {
                if ( rc_intersect(area, &box) )
                        disp_mesag(message, &box);
                wind_get(appl_whndl, WF_NEXTXYWH,
                        &box.g_x, &box.g_y, &box.g_w, &box.g_h);
        }
        graf_mouse(M_ON, 0L);
}
```

Listing 4.5: *HELLO Display Routines*

The hello_init( ) routine calls the wdw_size( ) routine to determine the appropriate window size and initial location to display the text message. The call to **vst_height( )** notifies the VDI to use a character set with a height equal to the type_size value. Since this value was the system font height returned from the **v_opnvwk( )** call, the VDI uses the system font. The remaining four parameters contain the character cell sizing information, which is used to calculate the internal window work area size, as seen in lines 15 and 16. Once the desired internal window size has been calculated, the external window size is calculated by calling **wind_calc( )** with the window's features (the internal window size plus border). This information, which is stored in the box structure, is used to size and locate the window when the window is opened in the initialization routine.

disp_message( ) actually displays the message inside the window, sets a number of device characteristics, and finally outputs the text. Since device characteristics never change throughout the course of the HELLO application, it might have been more efficient to make these calls in the initialization routine (or in wdw_size( ), as was done with the call to **vst_height( )**). However, the technique of grouping VDI attribute settings with VDI output calls makes a program easier to understand. This grouping technique is also a good idea in case the program is ever modified to display other kinds of information using VDI calls, in which case all of the relevant VDI calls will be grouped together.

The call to set_clip( ) on line 32 defines the clipping region of the screen. The clipping function makes it much easier to display information in windows, since you can clip a portion of your display in case another window overlaps yours. This very powerful technique is used again in the DEMO program presented in Chapter 5.

In lines 33 through 36, the background of the window is set to white. The local routine grect_to_array( ) changes the coordinate system from the GRECT rectangular specification (top-left x and y, with width and height) to the array specification used by the VDI (top-left x and y, bottom-right x and y).

It may seem unfortunate that the AES uses a rectangular specification method that must be regularly converted to the array specification used by the VDI. The main reason for this is that the AES uses mostly rectangular objects (like windows and forms), while the VDI was built to handle general-purpose polygonal shapes.

The next three lines (38 through 40) set the color and writing mode of the text. The mode used here is replace (MD_REPLACE) mode, which draws over whatever is currently on the screen. These modes were covered in Chapter 3's discussion of the VDI control functions.

Next, a while loop at line 42 calls the **v_gtext**( ) routine to display each line of text. The final call to set_clip( ) turns off clipping in order to spare later VDI calls the clipping overhead.

The do_redraw( ) routine illustrates how your GEM application must draw its information in order to peacefully coexist with many other windows. First, **graf_mouse**( ) removes the mouse form (the cursor) from the screen. The mouse form is displayed by saving a portion of the screen underneath the mouse form. Thus, if you don't turn off the mouse and you overwrite the mouse form area, the previous contents of the saved area are restored as soon as the mouse is moved, which leaves unpredictable results on the screen.

Next, **wind_get**( ) returns the start of a rectangle list, where each rectangle in the list defines a clipping area and is handed to the disp_mesag( ) routine. A null rectangle (whose width or height is 0) defines the end of the rectangle list. As we mentioned in Chapter 2, the rectangle list is how GEM supports overlapping windows. By dividing the screen into a series of rectangles, GEM can specify to each application the exact part(s) of the screen which need to be drawn into. The program uses the rectangle as a clipping region and redraws the entire window on top of each clipping rectangle, so that only the information inside the clipping region reaches the screen.

GENERAL-PURPOSE ROUTINES

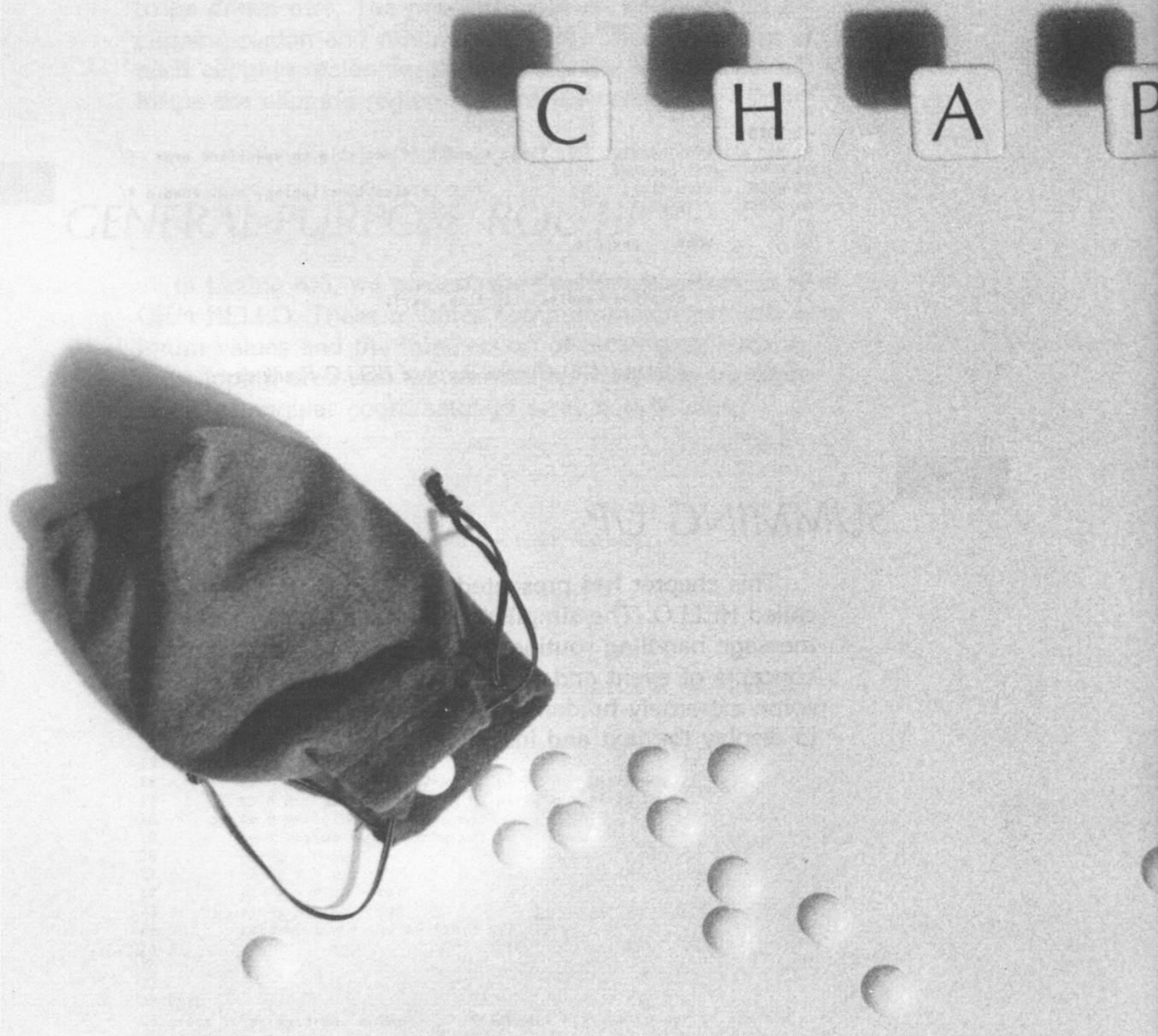
In Listing 4.6, we present the final few routines used in GEM HELLO. These routines compute maximum and minimum values and the intersection of rectangles, such as the clipping area and the work area. They also translate from rectangular coordinates to array coordinates.

```
/*
**      File:   hlocal.ci
**      Purpose: Most generalized low level routines.
*/

#define min(xx,yy) ((xx) < (yy) ? (xx) : (yy))
#define max(xx,yy) ((xx) > (yy) ? (xx) : (yy))


WORD
rc_intersect(p1, p2)            /* Compute intersection of two rectangles */
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) );
}


VOID
grect_to_array(area, array)     /* Convert x,y,w,h to upr lt x,y and     */
GRECT   *area;                  /*                lwr rt x,y             */
WORD    *array;
{
        *array++ = area->g_x;
        *array++ = area->g_y;
        *array++ = area->g_x + area->g_w - 1;
        *array = area->g_y + area->g_h - 1;
}




VOID
set_clip(vdi_handle, clip_flag, s_area) /* Set clip to specified area  */
WORD    vdi_handle;
WORD    clip_flag;              /* 0 to disable clipping, o.w. enable */
GRECT   *s_area;
{
        WORD    pxy[4];

        grect_to_array(s_area, pxy);
        vs_clip(vdi_handle, clip_flag, pxy);
}
```

Listing 4.6: *General-Purpose HELLO Routines*

SUMMING UP

This chapter has presented a simple GEM application called HELLO. The simplistic event handler in hello( ) and message handling routine hndl_mesag( ) represent the basic concepts of event and window handling. We also provided some extremely fundamental examples of using the VDI to display the text and inside of the window.


C H A P

T E R

5

GEM DEMO

In this chapter we examine in detail the sample program called DEMO in the DRI GEM Developer's Kit. The version that we use here is a reorganized version of the one found in the Kit, but it is still essentially the same. Our version is organized so that in the complete listing, which can be found in Appendix D, each function after GEMAIN( ) is listed alphabetically, which makes it easier to find any particular function. In addition, we have added commentary and cleaned up the code. By studying DEMO, you will learn how to use menus and dialogs, how to save and restore the screen, and how to process user input.

For reasons of brevity, the listings that appear in this chapter do not contain any of the comments that appear in the actual source. That source—the complete listing for DEMO—appears in Appendix D. We hope that our discussion alongside the code provides enough insight to compensate for the absence of comments in the listings themselves. To make the text's analysis of different portions of the program easier to follow, we have broken the listings of some of the longer functions into several pieces. In this way, the discussion and the listing can be kept close together in the chapter, which minimizes the number of times that you have to turn back to a listing as you read. Finally, to make the chapter simpler and easier to read, we also decided not to show all the routines that make up DEMO, but instead to concentrate on the most important ones. Thus, the listings in this chapter represent a selection from the complete program shown in Appendix D.

WHAT DEMO DOES

DEMO is a program that permits the user to draw (or doodle) on the screen using the mouse as a drawing instrument. The user depresses the leftmost mouse button, and holding it down, moves the mouse around the work area of the DEMO window. As the mouse moves, DEMO traces a line. DEMO is thus a simple drawing program—in effect, a simpler version of a sophisticated drawing program like GEM Draw.

DEMO uses menus and dialogs to provide additional services such as

- Saving the current screen to a disk file.
- Recalling a previously drawn doodle.
- Erasing the entire window.
- Changing line thickness.
- Changing from pencil to eraser mode so that the user can erase a portion of the drawing.

All in all, DEMO presents a consistent and complete model of pencil and paper.

DEMO START UP

As in GEM HELLO, the main routine is called GEMAIN( ) to avoid linking in all the normal C compiler run-time library overhead. The DEMO GEMAIN( ) shown in Listing 5.1 is very simple. It calls only three other routines: demo_init( ), demo( ), and demo_term( ).

```
GEMAIN()
{
        WORD    term_type;

        if (!(term_type = demo_init()))  demo();
        demo_term(term_type);
}
```

Listing 5.1: *GEMAIN()*

The initialization routine, demo_init( ), returns certain values that indicate how far the demo_init( ) routine proceeded. Various error conditions can force demo_init( ) to stop and return to GEMAIN( ) with DEMO only partly initialized. Depending on the return code (shown in Table 5.1), GEMAIN( ) either continues and invokes the main body of the DEMO code, or terminates.

demo_init( ) **Return Value**	**Meaning**
0	Normal initialization (no errors).
1	Could not load .RSC file, or could not open virtual workstation.
2	Could not allocate enough contiguous memory for screen buffer.
3	Could not create DEMO window because there were no more available windows.
4	Could not reserve any AES resources.

Table 5.1: *Meaning of Initialization Return Codes*

Let's take a quick look at the termination procedure.

DEMO TERMINATION

Depending on how far along the initialization proceeded before either encountering an error or completing demo_init( ), demo_term( ) needs to release any allocated resources that it obtained in demo_init( ). For instance, if demo_init( ) indicates that the resource file could not be loaded (return code = 1), the AES routines **appl_init**( ) and

wind_update( ) have been invoked. Before DEMO exits, the window must be unlocked and the resources allocated by **appl_init**( ) must be freed, which is exactly what demo_term( ) does. Thus, demo_term( ) makes sure that the GEM environment is all cleaned up before DEMO exits.

Listing 5.2 shows the code for demo_term( ).

```
demo_term(term_type)
WORD    term_type;
{
        WORD x, y, w, h;

        switch (term_type)      /* NOTE: all cases fall through          */
        {
          case (0):             /* Normal termination. */

                wind_get(demo_whndl, WF_CXYWH, &x, &y, &w, &h);
                wind_close(demo_whndl);
                graf_shrinkbox(full_width/2, full_hite/2, 21, 21,
                        x, y, w, h);
                wind_delete(demo_whndl);

          case (3):             /* No more windows available.  */

                menu_bar(0x0L, FALSE);
                dos_free(draw_mfdb.mp);

          case (2):             /* Couldn't open device.  */

                v_clsvwk( vdi_handle );

          case (1):             /* Couldn't find RSC file. */

                wind_update(END_UPDATE);
                appl_exit();

          case (4):             /* Error on appl_init(). */

                break;
        }
}
```

Listing 5.2: *DEMO Termination*

DEMO INITIALIZATION

demo_init( ) is more extensive than the initialization routine for GEM HELLO, because DEMO uses a number of

GEM capabilities that weren't used by GEM HELLO. Let's go through this routine, which is shown in Listing 5.3, step-by-step.

```
WORD demo_init() {

        WORD    work_in[11];
        WORD    i;

        if (appl_init() == -1) return(4);
        wind_update(BEG_UPDATE);
        graf_mouse(HOUR_GLASS, 0x0L);

        if (!rsrc_load( ADDR("DEMO.RSC"))){
        graf_mouse(ARROW, 0x0L);
        form_alert(1,
            ADDR("[3][Fatal Error !|DEMO.RSC|File Not Found][ Abort ]"));
        return(1);
        }

        vdi_handle=graf_handle(&char_width,&char_hite,&box_width,&box_hite);

        for (i=0; i<10; i++) {
                work_in[i]=1;
        }
        work_in[10]=2;                  /* Use RC coordinates */

        v_opnvwk(work_in,&vdi_handle,work_out);
        if (vdi_handle == 0) return(1);

        draw_mfdb.fwp = work_out[0] + 1;/* screen width in pixels */
        draw_mfdb.fh = work_out[1] + 1; /* screen height in pixels */
        scrn_xsize = work_out[3];       /* width (raster) aspect ratio */
        scrn_ysize = work_out[4];       /* height (raster) aspect ratio */
        char_fine = work_out[46];       /* min char height ptsout(1) */
        char_medium = work_out[48];     /* max char height ptsout(3) */
        char_broad = char_medium * 2;
```

Listing 5.3: *DEMO Initialization, Part 1*

A Walk Through demo_init()

The first three statements (the **appl_init()**, **wind_update()**, and **graf_mouse()** calls) allocate AES resources, obtain permission to update the screen, and change the mouse cursor form to an hourglass, respectively. All these functions have been dealt with before. As we discussed in Chapter 2, the call to **rsrc_load()** loads all the object-oriented data for DEMO into memory. (We will spend much of this chapter talking about objects later on.) **rsrc_load()** loads the object tree for the menu bar at the top of the DEMO window and all the associated submenus and dialogs. If an error occurs in loading this file, DEMO displays an alert telling the user that it could not find the resource file (.RSC) for DEMO, and demo_init() returns.

CHANGES IN HOW A GEM MENU BAR LOOKS

Because of the changes to GEM in the last few months, the appearance of the menu bar depends on which version of GEM you are running. Thus, in the old GEM (version 1), DEMO has three menus: Desk, File, and Options, each of which have submenus. The order in which they appear in GEM version 1 is as in Figure 5.1. However, when DEMO runs on the later version of GEM (version 2), you will see a File and Options menu, and to the far right a DEMO menu, as shown in Figure 5.2. The DEMO menu is the same as the prior version's Desk menu, only with a different main menu title. None of the data inside the object tree has changed. The only change is in the way the items are displayed, which means that old .RSC files are usable in the latest GEM.

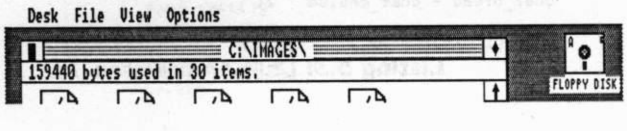


Figure 5.1: *Menu Bar for Version 1*

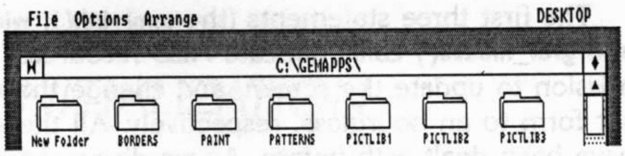


Figure 5.2: *Menu Bar for Version 2*

DEMO is now ready to open the virtual workstation. First DEMO gets the physical device handle (the call to **graf_handle**( )), and then it sets the other virtual workstation parameters to their defaults, mainly in order to use the raster coordinates. The call to **v_opnvwk**( ) changes the vdi_handle into the virtual workstation handle. **v_opnvwk**( ) returns other information about the physical screen that must be preserved in local variables since the next call to the VDI will overwrite those values.

Before we begin to discuss how demo_init( ) sets up the DEMO window, you need to understand how DEMO saves and restores the screen. To this end, we will first explain the data structures used by DEMO to handle the screen, and then we will show you how DEMO uses them to manipulate the image in the window.

DEMO Screen Handling

DEMO allows the user to size the DEMO window, as well as move the DEMO window around the physical screen. This means that as the user sizes the window smaller, DEMO must be able to change the view of what is displayed in the smaller window. The images that appear on the screen are stored in RAM. DEMO copies them to the screen and back again, thus managing the display. It is the responsibilty of all applications to handle all the memory management of their data structures in order to manage the screen.

Screen Data Structures

DEMO uses the six following data structures for window management:

— scrn_area and scrn_mfdb

— draw_area and draw_mfdb

—— work_area
—— save_area

scrn_mfdb and draw_mfdb are Memory Form Descriptor Block structures (see Chapter 3 for a discussion of MFDBs and how to use the VDI raster copy functions). The other four structures—draw_area, scrn_area, work_area and save_area—are GRECT structures (x, y, width, and height) as discussed in Chapter 4.

scrn_area and scrn_mfdb always refer to the actual physical screen (including borders) and are never changed after initialization. scrn_area and scrn_mfdb are used to provide the information about the physical screen size that is needed when the window gets changed: that is, either when the screen needs to be redrawn, when the user has finished drawing something on the screen, or when the user erases the entire screen. scrn_area is used to calculate the intersection of rectangles that have changed with the actual physical screen, while scrn_mfdb is used exclusively in the actual raster operation of copying from either the physical screen to the draw buffer or from the draw buffer to the screen.

draw_area refers to the GRECT description of the total screen area that is actually allocated for drawing upon, and draw_mfdb is the VDI description of that same area. Somewhat different information is included in draw_mfdb than in draw_area: for example, where the screen data is actually stored in RAM, the screen width both in pixels and words, and so on.

save_area is used to flip between full-screen and non-full-screen configurations. save_area contains the previous values of draw_area.

We'll talk about work_area shortly, but first let's discuss how DEMO computes the size of the screen buffer.

Calculating the Size of the Screen Buffer

The draw_area and draw_mfdb structures relate to the actual window work area in RAM that is allocated to DEMO. The call to the operating system memory allocation function (dos_alloc( ) in the middle of Listing 5.4) establishes the buffer of the required size by using the following equation:

width of physical screen ÷ 8 × height × number of planes

The width is divided by 8 because it is measured in pixel units, and dos_alloc( ) uses bytes.

```
vq_extnd(vdi_handle, 1, work_out);
draw_mfdb.np = work_out[4];      /* number of planes */

draw_mfdb.fww = draw_mfdb.fwp>>4;
draw_mfdb.ff = 0;

buff_size = (LONG)(draw_mfdb.fwp>>3) *
            (LONG)draw_mfdb.fh *
            (LONG)draw_mfdb.np;
buff_location = draw_mfdb.mp  = dos_alloc(buff_size);

if (draw_mfdb.mp == 0) return(2);

scrn_area.g_x = 0;
scrn_area.g_y = 0;
scrn_area.g_w = draw_mfdb.fwp;
scrn_area.g_h = draw_mfdb.fh;
scrn_mfdb.mp = 0x0L;
```

Listing 5.4: *DEMO Initialization, Part 2*

dos_alloc( ) returns either a pointer to a contiguous area in RAM that is as large as requested or else an error code. The pointer is saved in the draw_mfdb structure, as well as the other descriptors of the screen. Notice that the size of this area is as large as the physical screen. As Figure 5.3 shows, once borders are included, the draw_area does not fit on top of the physical screen window area.

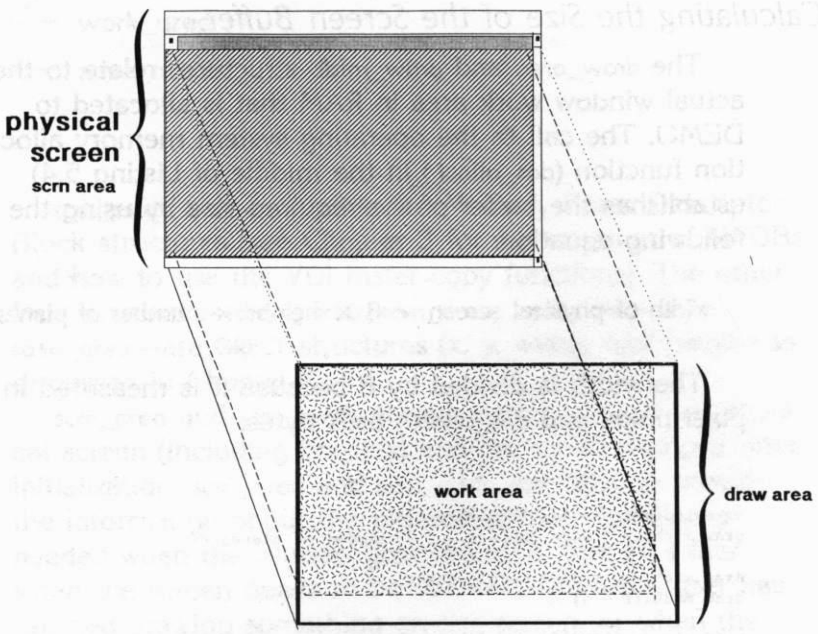


Figure 5.3: *Physical and Logical screen*

Relationships Between the Data Structures

In Figure 5.3, the physical screen is the background image which looks like a GEM window on a physical screen. Remember that scrn_area represents the physical screen including window borders, menu bars, and so on. What the dotted lines show is the mapping of the draw_area onto the physical screen (scrn_area). In effect, DEMO asks for enough space to allow a draw_area equal to the size of the physical screen, but GEM adds slider bars, a menu bar, and other window components, reducing the actual size of the picture that can appear on the screen at any one time. The slanted-line pattern indicates the greatest amount of draw_area that can appear, which also represents work_area (displayed as dashed lines). While there is more to draw_area that can appear on the screen in the window (work_area), to see it the user must use the sliders to scroll the window.

The relationship between the scrn and the draw structures is similar to that between the physical and the logical. scrn_mfdb and scrn_area describe the physical area, while draw_mfdb and draw_area describe the logical area. At any time during the execution of DEMO, a portion of the logical draw buffer is displayed on the physical screen described by the scrn structures. A good way to think of the relationship between these data structures is to imagine that a light is played through the draw_area and projects an image on the physical screen, much as a slide is projected on the movie screen. Ignoring the issue of focus, we can make the projected image smaller or larger by moving the projector backwards or forwards. Additionally we can project the image onto a small area of the screen; that is, we can make the image small and point the lens at a corner of the screen rather than projecting it onto the entire screen.

Now let's vary our analogy a little so that we can pay attention to focusing details. To keep the displayed image sharp and clear, let's keep the projector stationary. In order to get the effect of making the displayed image smaller and larger, imagine a rectangular gizmo in front of the lens that can expand along the width and the length of the rectangle. Thus by manipulating the sides of our elastic frame, we can size the image on the screen without losing the focus.

By the way, DEMO does not provide a "magnifying glass" feature. The image may change on the screen, but you cannot blow up a part of the image: it always appears at the same magnification level. In addition, DEMO also does not allow you to erase (undo) the last drawn line. One of the features of GEM Paint and Draw is an undo function, allowing the user to restore the picture as it appeared just before the last change.

Returning to our metaphor, scrn_mfdb and scrn_area describe the screen, draw_area and draw_mfdb describe the image in the projector, and work_area correlates to the elastic frame. draw_area defines the part of the image that can be displayed on the screen and that can fit through the frame of work_area. The image is controlled by the window sliders, whereas the size of the work area of the DEMO window is controlled by the size box and the move bar.

How work_area *and* draw_area *Work Together*

In a full-screen window configuration with no slider movement, the GRECTs of work_area and draw_area are the same. If the user moves the window down on the physical screen, then the coordinates for the area that is to be displayed on the screen (draw_area) haven't changed. The coordinates for work_area, however, now reflect the new position of the work area on the screen. The coordinates of the displayed image are the same. The image is clipped at the bottom of the window. If the user moves the slider positions, then the draw_area x and y coordinates change.

Most of the time the values of work_area and draw_area will be equal, if the user hasn't moved the window, resized it, nor changed the sliders—in other words, if the user keeps the initial configuration. work_area changes when the user moves or resizes the window. draw_area changes when the user wants to display a different part of the complete picture by moving the sliders.

Completing the Initialization: demo_init()

Now that we have explained the data structures that DEMO uses to manipulate the image on the screen, we can return to the demo_init() routine and finish initializing the DEMO environment as shown in Listing 5.5. What mainly remains to be done is to create, open, and display the DEMO window.

```
        rc_copy(&scrn_area, &draw_area);
        rast_op(0,&scrn_area,&scrn_mfdb,&draw_area,&draw_mfdb);

        pict_init();

        addr_msg = ADDR((BYTE *) &msg_buff[0]);

        wind_get(DESK, WF_WXYWH, &full_x, &full_y, &full_width, &full_hite);

        rsrc_gaddr(R_TREE, DEMOMENU, &addr_menu);

        menu_bar(addr_menu, TRUE);

        demo_whndl = wind_create(0x0fef, full_x - 1, full_y,
                                 full_width, full_hite);
        if (demo_whndl == -1)
        {                       /* No more windows available */

                form_alert(1, string_addr(DEMONWDW));
                return(3);
        }

        wind_set(demo_whndl, WF_NAME, ADDR(wdw_title), 0, 0);

        full_x = align_x(full_x);

        graf_growbox(full_width/2, full_hite/2, 21, 21,
                     full_x, full_y, full_width, full_hite);

        wind_open(demo_whndl,full_x,full_y,full_width,full_hite);

        set_work(TRUE);

        rc_copy (&draw_area, &save_area);

        graf_mouse(ARROW,0x0L); /* Restore arrow cursor */
        wind_update(END_UPDATE);/* Unlock update region */
        return(0);
}                               /* end demo_init */
```

Listing 5.5: *DEMO Initialization, Part 3*

We left demo_init( ) after it had reserved the necessary buffer space for the picture. demo_init( )'s next action is to clear the screen RAM area by using a VDI raster copying function (see Chapter 3). rast_op( ) converts the source and destination coordinates into the form that the VDI needs (refer to the discussion of the grect_to_array( ) function in Chapter 4) and then calls **vro_cpyfm( )** with a logic operation number of 0, which clears the destination bits. As we have said in Chapter 3, the logic operation number of 0 is like magic. It totally ignores whatever is in the source and just zeros out the destination.

pict_init( ) initializes the user-defined objects in the object tree. DEMO uses user-defined objects and bit images in the dialog to get the new pencil or eraser settings invoked from the Options menu. We will defer our discussion of this function until we cover objects in a later section of this chapter.

We are now at the point in demo_init( ) at which it builds the window and displays its menu. First, demo_init( ) gets the address of the object tree that was earlier loaded into memory through the **rsrc_load( )**. This object tree contains all the menu information for DEMO. **rsrc_gaddr( )** returns the address of that object tree in the parameter addr_menu. Once demo_init( ) has that address, it calls **menu_bar( )** to display the menu. Notice that a window handle is not specified. GEM only allows you one menu bar per screen, no matter how many windows you create.

The function **wind_get( )** with a value of WF_WXYWH (window work area) returns the coordinates of the indicated window, DESK, which is the largest window that can fit on the physical screen. DEMO uses the size of the work area of this window as the size of DEMO's work area. This is important because **wind_create( )** requires the size of the window at its fullest. You don't have to open the window at that size, but the AES needs to initialize certain internal variables.

DEMO builds a window that has everything except an information line: that is, the window includes a title bar, a move bar, full, close, and size boxes, horizontal and vertical sliders, and left, right, up, and down arrows. Please note that nothing has appeared on the screen yet but the cursor and the menu bar.

Finally, we are ready to open and display the initial DEMO window. The **grow_box( )** call is purely for effect. It draws an expanding box defined by the given parameters, giving the illusion that a tiny box on the screen grows into a large box. Unfortunately, you will not be able to see this effect on GEM version 2 or later versions since both this and the shrinking box effect have been removed from GEM (see Chapter 2's discussion of the Graphics Library). The entire call can be removed without damage.

The **wind_open**( ) call actually opens the window with the specified size, and draws the window control points. The elements that should so far have appeared on the screen are

- The DEMO menu bar.
- Hourglass mouse cursor.
- An expanding box in the middle of the physical screen (prior to version 2).
- Background dark hash pattern.
- The window control points, including the slider areas.

The screen shows nothing else. The next function call to set_work( ) sets the work coordinates and updates the slider area. The only time that the slider area is not updated is when the window is only moved, because the slider positions have not changed. Do not confuse resizing the window with window movement. The user moves a window by dragging the move bar around the screen, whereas resizing happens when the user drags on the size box in the lower right corner of the window.

When the window is opened, **wind_open**( ) sends a WM_REDRAW message to DEMO. It is not until DEMO receives and processes this message that the initialized work area of the window is displayed.

At this point the initialization of the DEMO window is complete. Figure 5.4 shows how the screen looks.

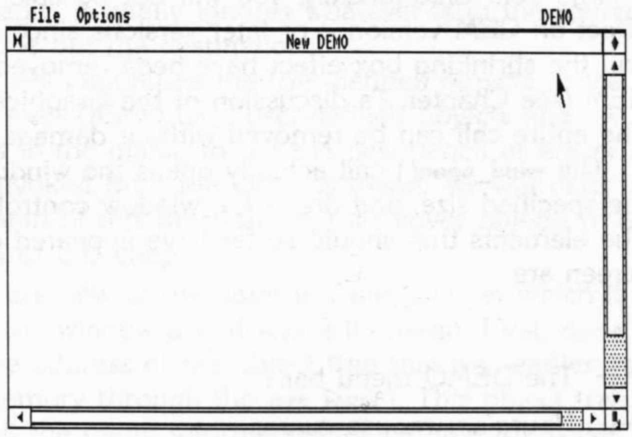


Figure 5.4: *The Initial DEMO Screen*

THE MAIN EVENT HANDLER: demo()

DEMO is now ready to begin processing of the user input. All control is passed through the main event handling routine, demo( ), which contains **evnt_multi( )**. As the user initiates some sort of GEM event with the mouse, the **evnt_multi**( ) in demo( ) catches it and activates the appropriate series of function calls. The main actions that DEMO waits for are the following:

— 1 The user selects a menu item.

— 2 The user clicks on the close or full box, moves or clicks on the sliders, moves the window, resizes the window, or used a desk accessory.

— 3 The user presses the far left button on the mouse.

— 4 The mouse moves out of the work area.

The Screen Manager sends DEMO a message when the first two actions occur. The demo( ) function, as shown in Listing 5.6, receives the message and passes control to the routine hndl_msg( ). As we will see, this routine sits on top of a very deep tree of routines that handle all the actions requested by the messages. demo( ) treats the last two items in the event list separately as a button and a mouse event, respectively, which are handled by the functions hndl_button( ) and hndl_mouse( ).

```
demo()
{
        WORD       junk;

        FOREVER
        {
                                /* Wait for mouse, message events only,
                                   left button goes down, mouse
                                   exits rectangle.
                                   addr_msg = address of message buffer  */

                ev_which = evnt_multi(MU_BUTTON | MU_MESAG | MU_M1,
                                0x02, 0x01, 0x01,
                                m_out,
                                (UWORD) work_area.g_x, (UWORD) work_area.g_y,
                                (UWORD) work_area.g_w, (UWORD) work_area.g_h,
                                0, 0, 0, 0, 0,
                                addr_msg, 0, 0,
                                &mousex, &mousey, &bstate, &kstate,
                                &junk, &bclicks);

                wind_update(BEG_UPDATE);

                if (ev_which & MU_MESAG) if (hndl_msg())  break;

                if (ev_which & MU_BUTTON) hndl_button();

                if (ev_which & MU_M1) hndl_mouse();
                wind_update(END_UPDATE);
        }
}
```

Listing 5.6: *DEMO Main Event Handler*

As demo( ) contains a forever loop, the only way that a user can break out of DEMO is to click on the Quit item in the File menu, or click on the close box of the window.

This information is sent back to demo( ) as a message that hndl_msg( ) then interprets, returning a true value to demo( ). None of the other user handlers called in demo( ) return anything.

Finally, notice that demo( ) locks the screen from being updated throughout the duration of handling the user input. DEMO must do this becuase it is going to change the screen upon processing the user input. Of course, you can move the **wind_update( )** calls down into the actual routines that actually make the changes, but placing the **wind_update( )** calls at this level is more than adequate. It is simple and does not require you to chase down an unmatched **wind_update( )** through the code, which is a real advantage when you consider how hard it is to debug code with unmatched stack operations (which is similar to turning on or off the update region).

Now let's examine the **evnt_multi( )** call. Only three events are of interest in this version of DEMO: messages, button action, and mouse movement out of the the mouse rectangle. The coordinates of the mouse rectangle correspond to the GRECT work_area. Any message is stored in the 16-byte message buffer pointed at by addr_msg, which was initialized in demo_init( ).

Mouse Rectangle Event Handler: hndl_mouse( )

Let's look at the simplest of the events that DEMO waits for: the mouse moves out of the rectangle defined by work_area. Quite simply, this event occurs when the user moves the mouse to the border area of the window, which activates the Screen Manager to handle all the user I/O. By convention, when the mouse is outside the work area of the application's window, the mouse cursor form is changed to an arrow. (This is only a convention and is not required by GEM.) Thus, DEMO changes the mouse cursor form to an arrow as shown in Listing 5.7. When the mouse comes back into the work area, DEMO changes it back to the cross hairs form (see Chapter 6 for more discussion of GEM conventions).

```
VOID hndl_mouse()
{
        if (m_out) graf_mouse(ARROW, 0x0L);
        else       graf_mouse(monumber, mofaddr);

        m_out = !m_out;
}
```

Listing 5.7: *DEMO Mouse Handler*

Mouse Button Event Handler: hndl_button()

Next let's examine the procedure that DEMO goes through to draw a picture on the screen. The depression of the leftmost mouse button while the mouse is in the work area triggers DEMO to begin tracing a line on the screen. If the mouse is in the border area of the screen, DEMO is asleep while the Screen Manager runs. Now the first thing that hndl_button() does is test to see if the current mouse coordinates are within the work area (see Listing 5.8). How can the mouse not be in the work area, and not be handled by the Screen Manager? What if the work area is smaller than the maximum window work area size?

```
VOID hndl_button()
{
        if ( (mousex >= work_area.g_x) && (mousey >= work_area.g_y) &&
             (mousex < (work_area.g_x + work_area.g_w)) &&
             (mousey < (work_area.g_y + work_area.g_h)))
                        draw_pencil(mousex, mousey);
}
```

Listing 5.8: *DEMO Mouse Button Handler*

Imagine that the user has resized the window to a size so that it sits in the center of the screen with lots of unused space between it and the physical screen edges. The mouse can be in that area of the screen where it is outside the work area, yet not in the border area where the Screen Manager will begin handling it. This no-man's-land is not available to DEMO as a drawing area. Furthermore, the Screen Manager does not process user input when the button is down before the mouse cursor moves over the menu bar or border area. (Try it. Hold the button down and run the mouse over the menu bar. No drop-down menus appear.) Finally, notice that DEMO is in a **wind_update()** region, preventing any other process from modifying the screen. It is for this reason that hndl_button() tests to see if the mouse is in the work area before handing over control to draw_pencil() to trace a figure.

Drawing a Picture: draw_pencil()

The first things that draw_pencil() does are to set the clipping rectangle to that of the work area (because the mouse can move outside the DEMO work area) and to draw a line (as shown in Listing 5.9). The clipping rectangle forces all drawing to stay within the confines of the work area. The code that is missing (see the complete listing in Appendix D) does some general maintenance things like setting the line style or eraser fill pattern. These tasks aren't important to our discussion of drawing on the screen, while the use of **evnt_multi()** is.

The **evnt_multi()** call is within a loop that tracks the mouse as the user moves it around the screen without releasing the button. This **evnt_multi()** call looks for one of the following events:

— The user releases the left mouse button (a button up *event*).

— A timer interrupt of 125 milliseconds.

— The mouse leaves the rectangle described by its last known position.

In Listing 5.10, we show the code that handles these three events.

```
WORD    draw_pencil(x, y)
UWORD   x, y;
{
        UWORD   pxy[4];
        WORD    done;
        UWORD   mflags;
        UWORD   locount, hicount;
        UWORD   ev_which, bbutton, kstate, kreturn, breturn;
        GRECT   tmp_area;

        set_clip(TRUE, &work_area);
        pxy[0] = x;             /* Save old mouse position */
        pxy[1] = y;

            ...REFER TO APPENDIX D FOR MISSING CODE...

        done = FALSE;
        while (!done){
            ev_which = evnt_multi(mflags,
                                  0x01, 0x01, 0x00,
                                  1, pxy[0], pxy[1], 1, 1,
                                  0, 0, 0, 0, 0,
                                  addr_msg, locount, hicount,
                                  &pxy[2], &pxy[3], &bbutton, &kstate,
                                  &kreturn, &breturn);
```

Listing 5.9: *DEMO Drawing Routine, Part 1*

```
        if (ev_which & MU_BUTTON)
        {
            if (!(mflags & MU_TIMER)) graf_mouse(M_OFF, 0x0L);
            if (demo_shade != PEN_ERASER)
                v_pline(vdi_handle, 2, (WORD *) pxy);
            else
                eraser((WORD) pxy[2], (WORD) pxy[3]);

            graf_mouse(M_ON, 0x0L);
            done = TRUE;
        }
        else
            if (ev_which & MU_TIMER)
            {
                graf_mouse(M_ON, 0x0L);
                mflags = MU_BUTTON | MU_M1;
                }
                else {
                        if (!(mflags & MU_TIMER))
                                graf_mouse(M_OFF, 0x0L);

                        if (demo_shade != PEN_ERASER)
                        {
                                v_pline(vdi_handle, 2, (WORD *) pxy);
                                mflags = MU_BUTTON|MU_M1|MU_TIMER;
                        }
                        else {
                                eraser((WORD) pxy[2], (WORD) pxy[3]);
                                graf_mouse(M_ON,0x0L);
                        }

                        pxy[0] = pxy[2];
                        pxy[1] = pxy[3];
                }
}                       /* ends while */
```

Listing 5.10: *DEMO Drawing Routine, Part 2*

First, if the user released the button, then draw_pencil() draws a line between the old mouse position and the new mouse position using the VDI line-drawing primitive, **v_pline()**. This is true if the cursor is really a pencil. If it is an eraser, then draw_pencil() draws instead a filled rectangle of the currently selected size with a fill color of white, thereby erasing anything on the screen at those coordinates.

When the user releases the left button, control must be returned to the main event handler (that is, to demo()), as this signals the end of drawing. The flag done controls the while loop in draw_pencil() and is set after draw_pencil() finishes tracing a line or erasing to ensure exit from the event loop of draw_pencil().

The second event that draw_pencil() waits for is a timer interrupt. This interrupt is used because DEMO turns the mouse cursor form, the cross hairs, off while the user traces on the DEMO work area (the call to **graf_mouse()** to turn off the cursor is in the complete listing of draw_pencil() in Appendix D). DEMO makes the mouse form invisible so that the mouse form doesn't get in the way of the drawing. However, it is difficult to draw without a cursor, because the user cannot see where on the screen the mouse is pointing to. So DEMO makes the mouse cursor form visible every 125 milliseconds or so. The mouse cursor form is turned off again as soon as one of the other events happen. This now-you-see-me-now-you-don't use of the cursor form is merely a stylistic device that the creators of DEMO chose. You must make your own decisions about what looks good in your own application.

Finally, the last event that draw_pencil( ) handles is mouse movement out of the specified rectangle (the 1 × 1 rectangle at the old mouse coordinates). The button is still down, but the user is moving the mouse, so that DEMO must show the new line being drawn. The smoothness of the line can be used to measure the speed at which mouse interrupts are being handled in GEM; the choppier the line, the more missed interrupts.

While the button is depressed, the event loop of draw_pencil( ) is still active. Thus, the new mouse position is saved as the old position and control is passed to the top of the event loop of evnt_multi( ).

Finishing the Picture

Only when the user releases the left button does control break out of the while loop of draw_pencil( ), prepatory to returning back to demo( ). Before it does, the environment must be cleaned up and the picture on the screen saved into the draw buffer. Listing 5.11 shows these actions.

The reason draw_pencil( ) turns off clipping is that we don't want to incur the overhead associated with clipping

```
        set_clip(FALSE, &work_area);

        graf_mouse(M_OFF, 0x0L);
        rast_op(3, &tmp_area, &scrn_mfdb, &draw_area, &draw_mfdb);
        graf_mouse(M_ON, 0x0L);
}
```

Listing 5.11: *DEMO Drawing Routine, Part 3*

if we don't need it. Clipping was necessary during the actual drawing, but DEMO will not need it until control returns to the drawing routine, draw_pencil( ).

Finally, the new data must be copied from the screen area to the draw buffer. The last several lines of code in draw_pencil( ) are important in illustrating the use of the various screen data structures. In this case, DEMO must copy the information drawn on the screen into the RAM buffer described by draw_area.

Notice that DEMO turns off the mouse cursor form before the actual raster copy of the screen. When the mouse form is on, the area of the screen that the form lies on top of is saved into a save area managed by the GEM VDI. This is exactly similar to what happens when a drop-down menu is enabled. If we copy the screen to a buffer, we want to copy everything that should be on the screen, and nothing else. Unfortunately the mouse form hides the underlying data, and it must therefore be turned off so that the entire image is present on the screen. Using **graf_mouse( )** to turn off the cursor restores the image that was underneath the cursor. When DEMO finishes copying, **graf_mouse( )** turns the mouse cursor back on.

The **rast_op( )** call shows a logic operation of 3, a source rectangle of the work_area, and the destination described by draw_area. All this means that we want to copy the source to the destination, replacing the values of the destination with the values of the source.

Message Handler: hndl_msg( )

We are now ready to discuss the remaining type of event that demo( ) waits for: message events. A great deal of DEMO concerns itself with processing messages. Most of the routines that hndl_msg( ) invokes are self-explanatory and have already been covered to some extent either earlier in the chapter (set_work( )) or in Chapter 2 (wind_set( ) and wind_calc( )). It will nevertheless be informative to discuss the application of these functions to the problem of managing user input. Before we do, it would be a good idea to review the material in the section on GEM message events in Chapter 2.

In Listing 5.12, notice first that the structure of hndl_msg( ) is that of a large switch statement, governed by the value of the message (msg_buff[0]). Unlike any of the other handlers that are called from demo( ), hndl_msg( ) returns a Boolean value. If this value is true, then the user either clicked on the close box or selected the "Quit" item in the File menu. In either case, DEMO terminates, leaving the switch loop either through a break statement, which signals that a false value was returned, or by returning directly from the switch with a true value.

```
BOOLEAN hndl_msg()
{
        WORD    wdw_hndl;
        GRECT   work;

        wdw_hndl = msg_buff[3];

        switch( msg_buff[0] )   /* Message type  */
        {
          case MN_SELECTED:       /* Mouse moved to menu */

                return(hndl_menu(wdw_hndl, msg_buff[4]));

          case WM_REDRAW:

                do_redraw(wdw_hndl, (GRECT *) &msg_buff[4]);
                break;

          case WM_TOPPED:         /* Place window on top */

                wind_set(wdw_hndl, WF_TOP, 0, 0, 0, 0);
                break;

          case WM_CLOSED:         /* Close the window */

                return(TRUE);

          case WM_FULLED:         /* Full button clicked */

                do_full(wdw_hndl);
                break;
```

Listing 5.12: *DEMO Message Handler, Part 1*

hndl_msg( ) examines ten messages, eight of which are window-related. One more is menu-related, and the final one is the screen redraw message. We will discuss those messages that effect a change in the window (including the redraw message) before looking at the menu selection message.

DEMO Window Management

Remember that all the window control points are outside the work area of the window and are handled by the Screen Manager. The Screen Manager sends messages to the application, indicating what the user wants to do. Some of the actions associated with the window-related messages include clicking on the window control points, slider movement, window movement, window resizing, and so on. First let's look at what happens when the user clicks on the full box.

DEMO Full Message Handler: do_full( )

When the user clicks on the full box, three things can happen: the window can be redrawn at its largest size, the window can be redrawn at its previous size, or nothing can happen. If the window is already at its largest size when the full box is clicked on, the window is redrawn to the immediately previous size. If the window is at some size other than full, it is redrawn to its largest size. Nothing happens when the window has never been sized differently from its initial state, and the window is at its largest size. Only one level of past window sizing information is kept, with the result that the application cannot select through a complete size history of the window. GEM maintains information about three states of the window sizes: current, previous, and full.

When the full box is clicked on, the Screen Manager sends a WM_FULLED message to the waiting application. In DEMO, hndl_msg( ) then invokes the do_full( ) function to process the request (Listing 5.13).

```
VOID    do_full(wh)
WORD    wh;
{
        GRECT   prev;
        GRECT   curr;
        GRECT   full;

        graf_mouse(M_OFF,0x0L);

        wind_get(wh, WF_FXYWH, &full.g_x, &full.g_y, &full.g_w, &full.g_h);
        wind_get(wh, WF_CXYWH, &curr.g_x, &curr.g_y, &curr.g_w, &curr.g_h);

        if ( rc_equal(&curr, &full) ){
                wind_get(wh, WF_PXYWH, &prev.g_x, &prev.g_y,
                          &prev.g_w, &prev.g_h);

                if (!( rc_equal(&prev, &full) )) {
                        graf_shrinkbox(prev.g_x, prev.g_y, prev.g_w, prev.g_h,
                                  full.g_x, full.g_y, full.g_w, full.g_h);

                        wind_set(wh, WF_CXYWH, prev.g_x, prev.g_y,
                                  prev.g_w, prev.g_h);

                        rc_copy(&save_area, &draw_area);
                        set_work(TRUE);
                }
        }
        else {                          /* Not full... */

                rc_copy(&draw_area, &save_area);
                graf_growbox(curr.g_x, curr.g_y, curr.g_w, curr.g_h,
                          full.g_x, full.g_y, full.g_w, full.g_h);

                wind_set(wh, WF_CXYWH, full.g_x, full.g_y, full.g_w, full.g_h);
                set_work(TRUE);
        }
        graf_mouse(M_ON,0x0L);
}
```

Listing 5.13: *DEMO Change to Full Screen and Back*

do_full( ) immediately obtains information about the previous and current window size and determines which one of two states the present window is in: full or less than full. If the window is currently full and the previous state was also full, then nothing needs to be done. However, if the current size is the same size as the full size and the previous size is not the full size, then DEMO must redraw the window in the smaller, previous size.

The use of the routine **graf_shrink box()** is again just for effect, and the routine will only show the shrinking box effect on the GEM version 1 system. It looks good to show the outline of shrinking box just before sizing the window down.

Now for the real magic, the call to **wind_set()**, which sets the values that define the current work area. In the process of changing the values, the AES also makes the necessary changes to the values associated with the previous window size (the full size never changes after window creation), but this does not change the window on the screen. Instead, the AES sends a WM_REDRAW message to any waiting application. It's important to remember that all GEM applications must be prepared to redraw the screen.

Although a redraw message has been sent, DEMO won't receive it until control returns to the topmost loop in demo( ). Before this happens, DEMO must update the value of draw_area to reflect the previous size, and update the work area. set_work( ) issues a **wind_get()** to obtain the size of the work area of the current window. But notice that the calls to **wind_set()** just prior to set_work( ) forced these values to reflect the new size. Thus set_work( ) updates the values in work_area with those of the newly sized window. (Note that the window still hasn't been redrawn to actually show this.)

Finally, control is passed back to hndl_msg( ), which returns to the main event loop, demo( ), in order to wait for any new messages, one of which will be the WM_REDRAW. Upon receiving this message, demo( ) passes control to hndl_msg( ), which calls do_redraw( ) to rewrite the screen.

Redrawing the DEMO Window: do_redraw( )

The problem at this point is that the image on the screen has changed and the window needs to be redrawn. We presented an overview of this process in the section on window redrawing in Chapter 2 and through a code example in Chapter 4. What happens is that the application walks down the list of rectangles that make up the image on the screen. For each rectangle, if there is any intersection between it and the area that has changed (that is, the *dirty rectangle*), then that intersection is redrawn. Listing 5.14 shows the DEMO redrawing routine.

```
VOID    do_redraw(wh, area)
WORD    wh;
GRECT   *area;
{
        GRECT   box;
        GRECT   dirty_source;

        graf_mouse(M_OFF, 0x0L);

        wind_get(wh, WF_FIRSTXYWH, &box.g_x, &box.g_y, &box.g_w, &box.g_h);
        while ( box.g_w && box.g_h ){
          if (rc_intersect(area, &box)){
                if (rc_intersect(&work_area, &box)){
                        dirty_source.g_x = (box.g_x - work_area.g_x) +
                                           draw_area.g_x;
                        dirty_source.g_y = (box.g_y - work_area.g_y) +
                                           draw_area.g_y;
                        dirty_source.g_w = box.g_w;
                        dirty_source.g_h = box.g_h;

                        rast_op(3, &dirty_source, &draw_mfdb,
                                  &box, &scrn_mfdb);
                }
          }
          wind_get(wh, WF_NEXTXYWH, &box.g_x, &box.g_y, &box.g_w, &box.g_h);
        }                       /* end while more rectangles... */
        graf_mouse(M_ON, 0x0L);
}
```

Listing 5.14: *DEMO Redraw Routine*

do_redraw( ) in DEMO is the routine that performs this function. It is only called from hndl_msg( ) when a WM_REDRAW message is received. The redraw operation continues while there is a rectangle in the list of rectangles for the screen. **wind_get**( ) returns width and height values of 0 to signal the end of the rectangle list. The do_redraw( ) function follows the procedure as already outlined in Chapters 2 and 4, but it may be difficult to follow the normalization of the dirty coordinates. Let's briefly look at this process.

The equation for the x and y points follows the same pattern as

source = dirty − work + draw

What is happening here is that the point is being normalized to the draw buffer. Thus, the first calculation (dirty − work) normalizes the point within the movement of the work_area on the screen. We know that the dirty rectangle must have a coordinate value that exceeds the corresponding work_area coordinate (for example, box.g_x > work_area.g_x). Subtracting the dirty point away from the work area point therefore gives an absolute value for the coordinate, which when added to the corresponding draw_area coordinate gives us the actual point in draw_area that describes the area that changed.

DEMO Slider Activity

The next three messages that hndl_msg( ) processes are WM_ARROWED, WM_HSLID, and WM_VSLID. First, let's look at what DEMO does when the user clicks on the slider area, which results in a WM_ARROWED message. Listing 5.15 shows the section of hndl_msg( ) that deals with this activity.

```
case WM_ARROWED:      /* Mouse touched slider area */

      switch(msg_buff[4])
      {
        case WA_UPPAGE:
              draw_area.g_y = max(draw_area.g_y-draw_area.g_h, 0);
              break;

        case WA_DNPAGE:
              draw_area.g_y += draw_area.g_h;
              break;
```

```
    case WA_UPLINE:
        draw_area.g_y = max(draw_area.g_y - YSCALE(16), 0);
        break;

    case WA_DNLINE:
        draw_area.g_y += YSCALE(16);
        break;

    case WA_LFPAGE:         /* Page left */
        draw_area.g_x = max(draw_area.g_x-draw_area.g_w, 0);
        break;

    case WA_RTPAGE:         /* Page right */
        draw_area.g_x += draw_area.g_w;
        break;

    case WA_LFLINE:         /* Column left */
        draw_area.g_x = max(draw_area.g_x - 16, 0);
        break;

    case WA_RTLINE:         /* Column Right */
        draw_area.g_x += 16;
        break;
  }

  set_work(TRUE);
  restore_work();
  break;
```

Listing 5.15: *DEMO Message Handler, Part 2*

Depending on what part of the slider area the user clicked on, different actions take place—for example, the actions of moving up or down a page or line. In this situation, the contents of msg_buff[4] provide the subcode necessary to identify what to do, which is either to add or subtract the appropriate value from the right point. For instance, if user wants to move the image up one page unit, then all that has to happen is that draw_area.g_y be shifted by a value equal to draw_area.g_h. Of course, the new value of draw_area.g_y cannot be less than 0.

Remember that as hndl_msg( ) shifts the values in draw_area DEMO eventually will have to show the new image area in the work area on the screen. The call to set_work( ) at the end of the switch in the WM_ARROWED section really only updates the position of the sliders, as work_area has not changed. It is not until the call to restore_work( ) that the screen gets changed.

Finally, let's examine how DEMO updates the slider position in the slider area. This position reflects how much of draw_area is visible. If the slider is in the center of the slider box, only the edges of draw_area are not visible. The AES returns the new relative position of the slider in msg_buff[4] with the messages WM_HSLID and WM_VSLID. msg_buff[4] is an integer value from 0 to 1000 that represents the position of the slider.

In our discussion of Listing 5.16, we will talk only about moving the horizontal slider since the explanation will be relevant to moving the vertical slider as well. When the WM_HSLID message is received, a new draw_area.g_x value must be calculated. UMUL_DIV(x,y,z) computes the value of (x*y)/z. DEMO calculates the new x position by adjusting the x position first within the physical screen (draw_mfdb.fwp – draw_area.g_w), multiplying that figure by the new relative slider position (msg_buff[4]), and then normalizing that number back within the range of 1000.

```
case WM_HSLID:        /* Horizontal slider position changed */

    draw_area.g_x = align_x(UMUL_DIV(draw_mfdb.fwp-draw_area.g_w,
                                      msg_buff[4], 1000));
    set_work(TRUE); /* Get new work area and update slider */
    restore_work();
    break;

case WM_VSLID:        /* Vertical slider position changed */

    draw_area.g_y = UMUL_DIV(draw_mfdb.fh - draw_area.g_h,
                             msg_buff[4],1000);
    set_work(TRUE);
    restore_work();
    break;
```

Listing 5.16: *DEMO Message Handler, Part 3*

We have finished discussing what DEMO does with respect to slider activity. The next two messages that hndl_msg( ) processes are WM_SIZED and WM_MOVED, which means that the user either clicked on the size button and resized the window or dragged the move bar around the screen, moving the window.

Sizing the DEMO Window

If the size button (or the move bar) has been clicked on, then the AES will draw the outline of a rubber box around the work area. The rubber box is visible as long as the user holds down the left button. The rubber box changes shape to reflect the movement of the mouse as the user resizes or moves the window. When the user releases the button, the Screen Manager sends either the WM_MOVED message or the WM_SIZED message to the owner of the window that was resized or moved, along with the new coordinates.

When a window is resized, the application must recompute the size of the window, whereas when a window is

```
        case WM_SIZED:      /* Size button clicked. */

            wind_calc(1, 0x0fef, msg_buff[4], msg_buff[5], msg_buff[6],
                      msg_buff[7], &work.g_x, &work.g_y, &work.g_w,
                      &work.g_h);
            work.g_x = align_x(work.g_x);
            work.g_w = align_x(work.g_w);

            wind_set(wdw_hndl, WF_CXYWH, msg_buff[4],
                     msg_buff[5], msg_buff[6], msg_buff[7]);

            set_work(TRUE);
            break;

        case WM_MOVED:      /* Window has been moved */

            msg_buff[4] = align_x(msg_buff[4]);
            wind_set(wdw_hndl, WF_CXYWH, align_x(msg_buff[4]) - 1,
                     msg_buff[5], msg_buff[6], msg_buff[7]);
            set_work(FALSE);
            break;

        }                   /* End switch */
    return(FALSE);
}
```

Listing 5.17: *DEMO Message Handler, Part 4*

moved, the application justs resets the work area coordinates. Listing 5.17 shows the section of hndl_msg( ) that handles window movement and resizing.

In the case of WM_SIZED, hndl_msg( ) must recalculate the size of the window using **wind_calc( )** instead of **wind_get( )**, because the new sizes haven't been set yet. **wind_calc( )** also computes the theoretical sizes from the number of components in the window (an input parameter to **wind_calc( )**). **wind_get( )** does not compute the sizes, but instead returns the values from internal variables.

The new coordinates (in msg_buff[4]) that are passed along with the WM_MOVED message include the border area. Thus the call to **wind_calc( )** asks for the new work area coordinates given the new border area coordinates. From these coordinates, the values for the current window configuration are set, which automatically updates the remaining two sets of window sizes (previous and full). In addition, as soon as the current window size is set, the AES sends a WM_REDRAW message to the owner of the window, resulting in the new work area being drawn on the screen. But before that message can be received and responded to, hndl_msg( ) updates the new value of the work area using the set_work( ) call both in the WM_SIZED and the WM_MOVED messages.

Notice that in Listing 5.17, the calls to set_work( ) in the WM_SIZED case and the WM_MOVED case have different input parameters. The parameter to set_work( ) relates to the updating of the slider area. When a window is resized, the sliders need to be updated too, because the amount of draw_area that is now visible in the work area has changed. When a window moves, the slider area does not need to change because the entire window moved, not some piece of it within the work area. Thus the sliders still show the correct ratio of visible to nonvisible image in the window. (Note that this is the only place in DEMO that set_work( ) is ever called with a value of false.)

USING OBJECTS IN DEMO

Now that we have discussed how DEMO uses windows, events, messages, and so on, we are ready to talk about objects, which provide the basis for a lot of the user's input into DEMO. The topic of objects will concern us for the rest of this chapter. In contrast to the more academic treatment of objects in Chapter 2, we are now going to see how objects get used within the context of a working application. DEMO uses objects extensively as the sole means of offering a choice of menus and dialogs to the user. This is hardly surprising since both menus and dialogs are another manifestation of objects. In fact, almost half of the AES libraries—the Object, Menu, Form, Resource, and File Selector Libraries—deal with objects in some way.

All the object trees that DEMO uses are contained within one resource file (DEMO.RSC) created by the Resource Construction Set. Appendix C shows you how to create some of the object trees for DEMO step-by-step. Appendix D shows you the complete listing of the DEMO resource file created by the RCS. Appendix D lists not only the object tree rs_object, but the bit image for the pencil and eraser and the strings that the object specification fields will point at. For now, we will begin by examining the structure of the object trees for DEMO.

DEMO Object Header File: demo.h

It is possible to associate defined names with each object in a tree. These names are another way of representing the indices of each object in the tree, but humans remember names better than numbers. Thus the RCS allows you to name an object, and then produces a .H file containing those names so that you can use the names in your program.

```
#define DEMOMENU        0       /* TREE */
#define DEMOINFO        8       /* OBJECT in TREE #0 */
#define DEMODESK        3       /* OBJECT in TREE #0 */
#define DEMOFILE        4       /* OBJECT in TREE #0 */
#define DEMOOPTS        5       /* OBJECT in TREE #0 */
#define DEMOLOAD        17      /* OBJECT in TREE #0 */
#define DEMOSAVE        18      /* OBJECT in TREE #0 */
#define DEMOSVAS        19      /* OBJECT in TREE #0 */
#define DEMOABAN        20      /* OBJECT in TREE #0 */
#define DEMOQUIT        21      /* OBJECT in TREE #0 */
#define DEMOPENS        23      /* OBJECT in TREE #0 */
#define DEMOERAP        25      /* OBJECT in TREE #0 */
#define DEMOINFD        1       /* TREE */
#define DEMOOK          6       /* OBJECT in TREE #1 */
#define DEMOPEND        3       /* TREE */
#define DEMOPSOK        10      /* OBJECT in TREE #3 */
#define DEMOCNCL        12      /* OBJECT in TREE #3 */
#define DEMOOVWR        0       /* STRING */
#define DEMONWDW        1       /* STRING */
#define DEMOSVAD        2       /* TREE */
#define DEMOSOK         2       /* OBJECT in TREE #2 */
#define DEMOSCNL        3       /* OBJECT in TREE #2 */
#define DEMONAME        4       /* OBJECT in TREE #2 */
#define DEMOIMG         3       /* OBJECT in TREE #1 */
#define DEMOPCLR        18      /* OBJECT in TREE #3 */
#define DEMOPFIN        3       /* OBJECT in TREE #3 */
#define DEMOPMED        4       /* OBJECT in TREE #3 */
#define DEMOPBRD        5       /* OBJECT in TREE #3 */
#define DEMOEFIN        6       /* OBJECT in TREE #3 */
#define DEMOEMED        7       /* OBJECT in TREE #3 */
#define DEMOEBRD        8       /* OBJECT in TREE #3 */
```

Listing 5.18: *DEMO Object Header File*

The file DEMO.H, shown in Listing 5.18, was created by the RCS and lists all the named objects in the resource tree for DEMO. Notice that there are four defined trees and two alerts (the alerts are really trees):

— 1 DEMOMENU (tree 0).

— 2 DEMOINFD (tree 1).

— 3 DEMOSVAD (tree 2).

— 4 DEMOPEND (tree 3).

— 5 Alert for overwriting an existing file.
— 6 Alert for using all the windows.

Instead of looking at each tree, we will only concentrate on the main menu tree (DEMOMENU) and the tree for the Pencil/Eraser Selection dialog (DEMOPEND). You will be able to understand the others after we finish with DEMOMENU and DEMOPEND.

A Detailed Look at the Object Trees

Tables 5.2, 5.3, 5.4, and 5.5 show the object trees for DEMO. We have only shown the first six elements of the actual object, leaving off the object specification field and the x, y, width, and height fields in order to allow for space in the tables to show the strings that the specification field might point to. Appendix B shows the full and unedited object tree listing. Please remember that while the object specification field may contain a pointer, until the program gets loaded into memory there is no way that the tree can contain an actual valid pointer. Therefore, some object specification fields may look funny.

Object	Next	Head	Tail	Type	Flags	State	Associated String
0	−1	1	6	G_IBOX	NONE	NORMAL	
1	6	2	2	G_BOX	NONE	NORMAL	
2	1	3	5	G_IBOX	NONE	NORMAL	
3	4	−1	−1	G_TITLE	NONE	NORMAL	Desk
4	5	−1	−1	G_TITLE	NONE	NORMAL	File
5	2	−1	−1	G_TITLE	NONE	NORMAL	Options
6	0	7	22	G_IBOX	NONE	NORMAL	
7	16	8	15	G_BOX	NONE	NORMAL	

Object	Next	Head	Tail	Type	Flags	State	Associated String
8	9	−1	−1	G_STRING	SELECT.	NORMAL	About GEM Demo . . .
9	10	−1	−1	G_STRING	NONE	DISABLED	-------------------
10	11	−1	−1	G_STRING	NONE	NORMAL	1
11	12	−1	−1	G_STRING	NONE	NORMAL	2
12	13	−1	−1	G_STRING	NONE	NORMAL	3
13	14	−1	−1	G_STRING	NONE	NORMAL	4
14	15	−1	−1	G_STRING	NONE	NORMAL	5
15	7	−1	−1	G_STRING	NONE	NORMAL	6
16	22	−1	−1	G_BOX	NONE	NORMAL	
17	18	−1	−1	G_STRING	SELECT.	NORMAL	Load
18	19	−1	−1	G_STRING	NONE	DISABLED	Save
19	20	−1	−1	G_STRING	SELECT.	NORMAL	Save as . . .
20	21	−1	−1	G_STRING	NONE	NORMAL	Abandon
21	16	−1	−1	G_STRING	SELECT.	NORMAL	Quit
22	6	23	25	G_BOX	NONE	NORMAL	
23	24	−1	−1	G_STRING	NONE	NORMAL	Pencil/Eraser Selection
24	25	−1	−1	G_STRING	NONE	DISABLED	--------------------
25	22	−1	−1	G_STRING	LASTOB	NORMAL	Erase Picture

Table 5.2: *DEMOMENU Object Tree*

Object	Next	Head	Tail	Type	Flags	State	Associated String
0	−1	1	4	G_IBOX	NONE	OUTLINED	
1	2	−1	−1	G_STRING	NONE	NORMAL	Save GEM Demo Pictures as
2	3	−1	−1	G_BUTTON	0x7	NORMAL	OK
3	4	−1	−1	G_BUTTON	0x5	NORMAL	Cancel
4	0	−1	−1	G_FTEXT	0x28	NORMAL	________.DOO

Table 5.3: *DEMOSVAD Object Tree*

Object	Next	Head	Tail	Type	Flags	State	Associated String
0	−1	1	11	G_BOX	NONE	OUTLINED	
1	2	−1	−1	G_STRING	NONE	NORMAL	GEM Demo
2	1	−1	−1	G_STRING	NONE	NORMAL	GEM Sample Application
3	4	−1	−1	G_IMAGE	NONE	NORMAL	(DRI Logo)
4	5	−1	−1	G_STRING	NONE	NORMAL	Authors
5	6	−1	−1	G_STRING	NONE	NORMAL	-------
6	7	−1	−1	G_BUTTON	0x7	NORMAL	OK
7	8	−1	−1	G_STRING	NONE	NORMAL	Tom Rolander
8	9	−1	−1	G_STRING	NONE	NORMAL	Tim Oren
9	10	−1	−1	G_STRING	NONE	NORMAL	Phillip Balma
10	11	−1	−1	G_STRING	NONE	NORMAL	Version 1.2, February 1, 1986
11	12	−1	−1	G_STRING	LASTOB	NORMAL	Digital Research, Inc.

Table 5.4: *DEMOINFD Object Tree*

Object	Next	Head	Tail	Type	Flags	State	Associated String
0	−1	1	13	G_BOX	NONE	OUTLINED	
1	2	−1	−1	G_STRING	NONE	NORMAL	GEM Demo Pencil/Eraser Selec.
2	9	3	8	G_BOX	NONE	NORMAL	
3	4	−1	−1	G_IMAGE	0x11	NORMAL	
4	5	−1	−1	G_IMAGE	0x11	NORMAL	
5	6	−1	−1	G_IMAGE	0x11	NORMAL	
6	7	−1	−1	G_IMAGE	0x11	NORMAL	
7	8	−1	−1	G_IMAGE	0x11	NORMAL	
8	2	−1	−1	G_IMAGE	0x11	NORMAL	
9	10	−1	−1	G_STRING	NONE	NORMAL	Pens:
10	11	−1	−1	G_BUTTON	0x7	NORMAL	OK
11	12	−1	−1	G_STRING	NONE	NORMAL	Erasers:
12	13	−1	−1	G_BUTTON	0x5	NORMAL	Cancel
13	14	−1	−1	G_IBOX	NONE	NORMAL	
14	15	−1	−1	G_STRING	NONE	NORMAL	Pen Colors:
15	18	16	17	G_IBOX	NONE	NORMAL	
16	17	−1	−1	0x11B	TOUCHEXIT	NORMAL	
17	15	−1	−1	0x21B	TOUCHEXIT	NORMAL	
18	23	19	22	G_IBOX	NONE	NORMAL	
19	20	−1	−1	0x31B	TOUCHEXIT	NORMAL	
20	21	−1	−1	0x31B	TOUCHEXIT	NORMAL	
21	22	−1	−1	0x31B	TOUCHEXIT	NORMAL	
22	18	−1	−1	0x31B	TOUCHEXIT	NORMAL	
23	24	−1	−1	G_BOXCHAR	NONE	NORMAL	
24	25	−1	−1	G_STRING	LASTOB	NORMAL	Selected:

Table 5.5: *DEMOPEND Object Tree*

It's important to note that the Resource Construction Set builds the tree in the order that you process the objects, which may not necessarily be the way that they appear on the display.

THE USEFULNESS OF THE RCS

Let's reemphasize the utility of the RCS. While some of you might not think it an arduous task to build these trees, consider the level of detail involved in just placing the object on the actual screen (these coordinates are the final four words in the object descriptor). The RCS allows you to move the object around on the screen and expand it or contract it until you are satisfied with its appearance on the screen. Then it automatically fills in the x, y, width, and height values associated with that object with respect to the root object in the tree. This feature alone is invaluable, as without it you would have to calculate by hand the coordinates of the final location of the root of the object tree. Then you would have to go through each of the objects in the tree and set their coordinates relative to the coordinates of the root.

DEMOMENU *Object Tree*

First let's look at the simpler of the two trees we will be discussing, DEMOMENU. It has 26 objects, 19 of which are strings of some sort (either titles or strings). Figure 5.5 shows the way the tree looks with respect to pointers to objects. The text ("Desk," "Quit," and so on) is placed there to give you a reference point. To save space, we have left out the pointer to the next sibling between the G_STRING objects themselves (object 3 points to object 4, object 4 points to object 5, and so on). The objects that touch in Figure 5.5 indicate a sibling relationship.

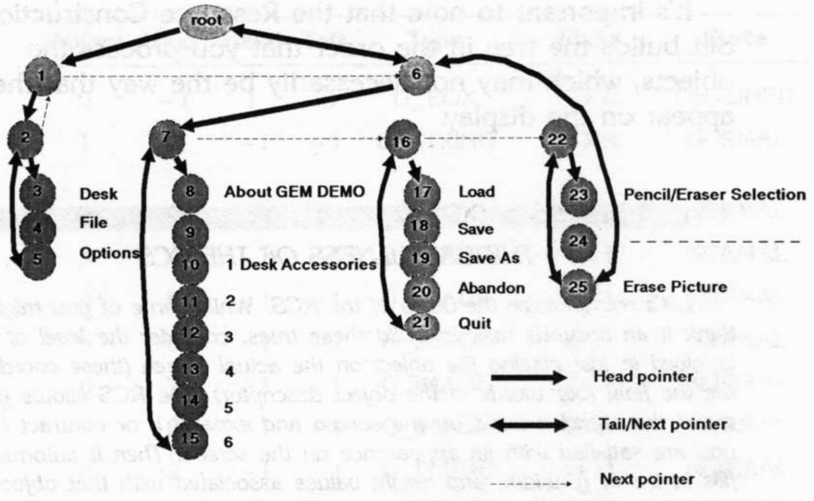


Figure 5.5: *DEMOMENU Object Tree I*

Referring to Table 5.2 again, notice that all the object types are defined by GEM (either G_BOX, G_IBOX, G_TITLE, or G_STRING). In the DEMOPEND tree, we will see some user-defined objects. If the flags field shows SELECTABLE, that means the object can be selected. Notice that object 18's state is DISABLED. This means that upon the initial invocation of this menu, the user cannot save the doodle into a known file. This makes sense as the user has not opened a file and identified it to DEMO, so DEMO has no name under which to save it. When the user either opens a file or saves the doodle using the "Save As . . ." menu item, DEMO has a file name to use as the name of the file in which to save the doodle. Of course, as soon as that happens, the state of this object will change from DISABLED, and its flags field will become SELECTABLE. We want to make it clear that GEM does not do any of this for you automatically. The application must perform all this object manipulation and changing of states itself.

A lot of the code of DEMO concerns itself with turning on or off various fields of DEMO's object trees. The Screen Manager selects the item in the menu, for instance, and it is the application's job to "deselect" it, since you certainly don't want to the user to see the last selected item highlighted the next time the menu drops down.

This brings up the point that the ability to select something in GEM is indicated by the appearance of the item. If it is not selectable, then it appears in a lighter shade than the selectable counterpart. This use of highlighting is a GEM convention (see Chapter 6 for more information about conventions).

While Figure 5.5 does not show the type of object, in Figure 5.6 the object tree is restructured to show the relationship between the types. The boxes formed by dashed lines are G_IBOXs, and solid-lined boxes are G_BOXs. The small numbers indicate the object index.

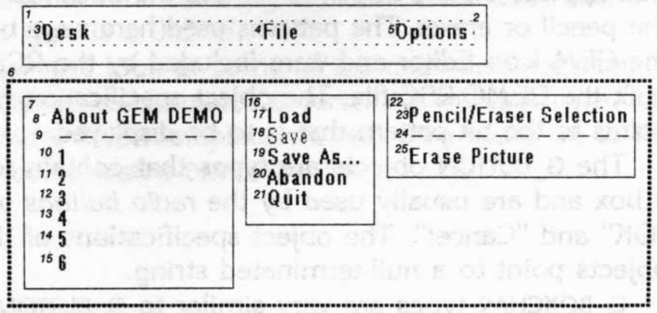


Figure 5.6: *DEMOMENU Object Tree (II)*

DEMOPEND *Object Tree*

Refer to both Table 5.5 and Figure 5.7 to see the last example of an object tree that we will study. The dialog DEMOPEND, shown in Figure 5.7, is important because of the way in which DEMO manipulates the elements of the tree and also because it is an example of user-defined objects.

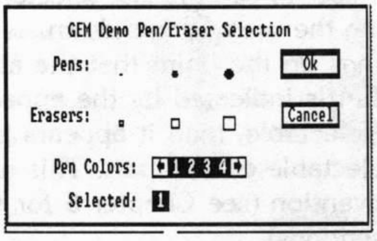


Figure 5.7: *Pencil/Eraser Selection Dialog*

DEMOPEND uses some different object types than DEMOMENU: G_IMAGE, G_BUTTON, G_BOXCHAR, and three user-defined types. The G_IMAGE objects refer to the bit pattern that appears on the screen to denote the thickness of either the pencil or eraser. The patterns used here were built using the GEM Icon Editor and were included by the RCS when it built the DEMO.RSC file. The object specification field points to the bit pattern that is to be displayed.

The G_BUTTON objects are types that contain text within a box and are usually used by the *radio buttons* to display "OK" and "Cancel". The object specifications of these objects point to a null-terminated string.

G_BOXCHAR types are very similar to G_BUTTON, except that G_BOXCHAR objects contain only one character, and the object specification contains the character as well as information about the color of the character, instead of a pointer to text.

The three user-defined objects refer to the color selector of the dialog. Here the forward and backward arrows, as well as the four visible color choices, are user-defined. By that we mean that they cannot be manipulated directly by the AES, but are controlled by the application. In this case, there are three things that happen when the user clicks on the color selector.

First, if the forward arrow is selected, then the four choices must be moved ahead by one item. Thus, a field of showing "2345" shifts to "3456". As there are only 15 colors to choose from, when the last color becomes visible in the selector, the next color to be displayed must be the first or "1". Thus, "CDEF" gets changed to "DEF1". The forward arrow is a user-defined object type of 0x11B.

Second, if the backward arrow is selected, then the logically similar thing must occur: "2345" gets changed to "1234". When the first color becomes visible, the next color to be displayed must be the last color, "F". Thus, "1234" gets changed to "F123". The backward arrow object type is a 0x21B.

Third, if a particular visible color gets clicked on, then object 23 (see Table 5.5), which is a G_BOXCHAR that shows which color is currently selected, must be changed to show the new currently selected color. This user-defined object type is a 0x31B.

Now let's look at some of the code and see exactly how DEMO manipulates objects. We will first consider the DEMOMENU tree manipluation, and then move on to the far more complicated DEMOPEND tree.

A Walk Through DEMOMENU

The object handler code starts off innocently enough. In a very top-down fashion, a MN_SELECTED message is sent from the Screen Manager to DEMO that something happened in the menu bar area of the screen. As shown in Listing 5.12, hndl_msg( ) passes the message on to hndl_menu( ), which is shown in Listing 5.19.

```
WORD    hndl_menu(title, item)
WORD    title, item;
{
        LONG    tree;
        GRECT   box;

        graf_mouse(ARROW, 0x0L);
        switch (title) {
          case DEMODESK:
                if (item == DEMOINFO) {
                        objc_xywh(addr_menu, DEMODESK, &box);
                        rsrc_gaddr(R_TREE, DEMOINFD, &tree);
                        hndl_dial(tree, 0, box.g_x, box.g_y, box.g_w, box.g_h);
                        desel_obj(tree, DEMOOK);
                }
                break;

          ...REFER TO APPENDIX D FOR MISSING CODE ...

          case DEMOOPTS:
                switch (item) {
                  case DEMOPENS:
                        do_penselect();
                        break;
                  case DEMOERAP:

                        rast_op(0, &scrn_area, &scrn_mfdb,
                                &scrn_area, &draw_mfdb);
                        restore_work();

                        break;
                }
        }
        menu_tnormal(addr_menu,title,TRUE);
        graf_mouse(monumber, mofaddr);
        return (FALSE);
}
```

Listing 5.19: *DEMO Menu Handler*

For the sake of illustration, let's suppose that the user clicked on the "About GEM Demo . . ." item in the DEMO-DESK menu. (This menu will be in a different place on the screen and titled differently depending on the version of GEM that the user is running. See Figures 5.1 and 5.2.) DEMO must now display the information in the DEMOINFD dialog.

Please be careful not to get confused between the the DEMOINFD tree and the DEMOINFO menu item in the DEMO-DESK menu. It is easy to misread the "D" for an "O". The

DEMOINFD tree is the name of the dialog that is displayed when the DEMOINFO menu item is selected.

There are two values passed to hndl_menu( ): the object index of the menu that was selected (DEMODESK), and the object index of the item that was selected (DEMOINFO) in that menu.

First hndl_menu( ) changes the mouse cursor to an arrow form, in keeping with the GEM convention of using an arrow mouse form when dealing with the application's resources. Using a switch statement, hndl_menu decodes the message. In our example, we will look only at the first part of the switch, corresponding to the DEMODESK case.

There is only one selectable item in this menu that concerns DEMO: DEMOINFO. The other items refer to any desk accessories that get loaded. An application does not have to worry about managing any of the desk accessories resources, so DEMO just ignores any other item selection.

The next four lines of code reveal the basic set of activities that must occur when a dialog gets put on the screen. (Note that objc_xywh( ) is *not* an AES function. It has an unfortunate name in that while it is descriptive, it looks too much like the C binding names of the AES Object Library.) objc_xywh( ) returns the coordinates of the DEMODESK tree relative to the screen. The only reason for getting this information is to have a starting point when we draw an expanding/shrinking box later on. The entire call can be excised without danger.

The call to **rsrc_gaddr( )** returns the address of the DEMOINFD tree in the resource tree. DEMO uses this information in the next routine that is called, hndl_dial( ).

Dialog Handler: hndl_dial( )

hndl_dial( ), shown in Listing 5.20, is called by all the routines in DEMO that use dialogs: hndl_menu( ), dial_name( ), and do_penselect( ). The way that DEMO displays a dialog demonstrates all the steps necessary for handling objects, as well as some optional steps to make things look good.

There have been some changes to the **form_dial( )** function in version 2. As mentioned in Chapter 2's discussion of the Form Library, **form_dial( )** no longer draws an expanding or shrinking box. Neither does it reserve or release space. In fact, it has become quite unclear what purpose **form_dial( )** serves in the new version of GEM.

```
WORD    hndl_dial(tree, def, x, y, w, h)
LONG    tree;
WORD    def;
WORD    x, y, w, h;
{
        WORD    xdial, ydial, wdial, hdial, exitobj;
        WORD    xtype;

        form_center(tree, &xdial, &ydial, &wdial, &hdial);
        form_dial(0, x, y, w, h, xdial, ydial, wdial, hdial);
        form_dial(1, x, y, w, h, xdial, ydial, wdial, hdial);

        objc_draw(tree, ROOT, MAX_DEPTH, xdial, ydial, wdial, hdial);

        FOREVER                 /* Get the user's input */
        {
                exitobj = form_do(tree, def) & 0x7FFF;
                xtype = LWGET(OB_TYPE(exitobj)) & 0xFF00;
                if (!xtype) break;
                xtend_do(tree, exitobj, xtype);
        }

        form_dial(2, x, y, w, h, xdial, ydial, wdial, hdial);
        form_dial(3, x, y, w, h, xdial, ydial, wdial, hdial);

        return (exitobj);
}
```

Listing 5.20: *DEMO Dialog Handler*

DEMO also centers the dialog and draws expanding/shrinking boxes (in version 1). The call to **form_center( )** gets the screen coordinates so that the DEMOINFD tree is displayed in the center of the screen.

The dialog still has not appeared on the screen. The call to **objc_draw( )** does the final drawing, and displays the dialog. This routine only displays as much of the tree as you specify in the third parameter (here MAX_DEPTH). None of the trees in DEMO exceed a depth of 10.

All the user interaction that is not controlled by the user-defined objects is handled by the **form_do( )**. (The source code for this routine is distributed with the Developer's Kit.) Because DEMOINFD only allows the user to select the OK button, **form_do( )** will not return until that has happened. The index of the selected object is the return value.

Let's defer most of the discussion about the code inside the forever loop of hndl_dial( ) until we discuss the user-defined objects. Right now, DEMO masks off the sign bit of the returned object index value, and then finds the type of selected object. All the predefined object types are masked off by the mask 0xFF00. Since the object type of the OK button in DEMOINFD is a G_BUTTON (0x001A), DEMO breaks out of the loop.

Now all that has to be done is to back out, restoring the environment as we go. hndl_dial( ) returns to the hndl_menu( ), which deselects the DEMOINFO item in the DEMODESK menu.

Handling User-Defined Objects

We have just seen the basic model of how object trees are handled, and how a user interacts with them. In some circumstances, you will need to use your own defined objects because GEM may not supply what you need. DEMO demonstrates such a case. In DEMO the color selection bar is not one of the tools supplied by GEM. Let's look at this user-defined object now.

User-Defined Color Selection

When hndl_menu( ) identifies that the object DEMOPENS was selected, it passes control to the routine do_penselect( ), which is shown in Listing 5.21. As in the previous section, DEMO obtains the coordinates of the DEMOPENS object relative to the screen, and uses them solely to draw an expanding/shrinking box for effect later on (in version 1 only). And again, **rsrc_gaddr()** returns the address in memory of the root of the DEMOPEND tree.

```
VOID    do_penselect()          /* use dialog box to input selection    */
{                               /*  of specified pen/eraser             */

        WORD    exit_obj, psel_obj, color;
        LONG    tree, bind[2];
        GRECT   box;

        objc_xywh(addr_menu, DEMOPENS, &box);
        rsrc_gaddr(R_TREE, DEMOPEND, &tree);

        switch (demo_pen) {
                case PEN_FINE:
                 sel_obj(tree, (demo_shade != PEN_ERASER)?DEMOPFIN:DEMOEFIN);
                 break;
                case PEN_MEDIUM:
                 sel_obj(tree, (demo_shade != PEN_ERASER)?DEMOPMED:DEMOEMED);
                 break;
                case PEN_BROAD:
                 sel_obj(tree, (demo_shade != PEN_ERASER)?DEMOPBRD:DEMOEBRD);
                 break;
        }
        set_select(tree, DEMOPCLR, pen_shade - 1, bind, color_sel);

        exit_obj = hndl_dial(tree, 0, box.g_x, box.g_y, box.g_w, box.g_h);

        for (psel_obj = DEMOPFIN; psel_obj <= DEMOEBRD; psel_obj++)
                if (LWGET(OB_STATE(psel_obj)) & SELECTED)
                {
                        desel_obj(tree, psel_obj);
                        break;
                }
         ...REFER TO APPENDIX D FOR MISSING CODE...
```

Listing 5.21: *DEMO Pencil/Eraser Selection*

The next set of instructions essentially does the following:

— 1 Set the object state of the currently selected pencil or eraser shade to SELECTED.

— 2 Do the dialog.

—— 3 Clear the newly selected object (turn off the SELECTED state).

—— 4 Save the new input in nonobject variables.

DEMO seems to be turning things on, and then turning them off again, because of the data structures that are being used. Objects extract a toll in overhead, and that overhead is very evident here. Before do_penselect( ) executes the actual user input routine (in hndl_dial( ) and **form_do**( )) and changes the values of some the object variables, DEMO needs to have a known state to start from. All this model is doing is ensuring that we only have one selected object to choose from.

So let's go on to the set_select( ) function call, as this is critical for the proper function of the DEMOPEND dialog. Please refer to the declarations at the front of the complete listing of DEMO in Appendix D. Just before the GEMAIN( ) routine is a block of MFDBs for the mouse cursor form when the eraser is selected, and just before that is a LONG array called color_sel[ ]. These are the values of the numbers displayed within the color selector in DEMOPEND. If you look closely at them, you will see that the values obey the rules for values pointed at by the G_BOXCHAR. The high byte of the high word is the actual character value (in ASCII), and the low byte of the high word is the thickness of the borders. An 0xFF indicates the thickest inside border. The low word is the color word, as described in Figure 2.9. There are 16 values in color_sel. The first value is the actual number of colors contained in the array; thus, there are only 15 colors, 0x1 through 0xF.

Now look closely at set_select( ), shown in Listing 5.22. This procedure sets the colors into the DEMOPEND tree. DEMOPCLR is the head of the objects that comprised the color selector, and it is a G_IBOX object type. DEMO overwrites its object specification field with the address of the beginning of the color_sel array. set_select( ) also changes the object flag for DEMOPCLR to INDIRECT, indicating that the value in the object specification is a pointer to the actual object specification. In fact, there are two values that the object specification is pointing at: the original object specification and the address of the color array. Notice that set_select( ) stores these values in the local array bind[ ]. It does this so that these values are available for the duration of the operation of the dialog. The routine move_do( ) in fact uses the values in bind to shift the visible color selector.

```
VOID    set_select(tree, obj, init_no, bind, arry)
LONG    tree, bind[], arry[];
WORD    obj, init_no;
{
        WORD    n, nobj, cobj, count;

        indir_obj(tree, obj);
        bind[0] = LLGET(OB_SPEC(obj));
        LLSET(OB_SPEC(obj), ADDR(bind));
        bind[1] = ADDR(arry);

        n = (WORD) arry[0];
        count = 0;
        for (cobj = LWGET(OB_HEAD(obj)); cobj != obj;
            cobj = LWGET(OB_NEXT(cobj)))
        {
                indir_obj(tree, cobj);
                LLSET(OB_SPEC(cobj), ADDR( &arry[count + 1] ));
                count = (count + 1) % n;
        }
        nobj = LWGET(OB_NEXT(obj));
        indir_obj(tree, nobj);
        LLSET(OB_SPEC(nobj), ADDR( &arry[1 + init_no % n] ));
}
```

Listing 5.22: *DEMO Color Selection*

set_select( ) also sets the first four values of the user-defined objects (objects 19 through 22 in DEMOPEND). Again, the object specification of the user-defined objects is replaced by the address of the appropriate color in the array. The flags of each object are changed to reflect the indirect nature of the object specification. In this case, however, the pointer in the object specification field really does point to a data type that is expected and defines the character that appears in the selector.

Finally, set_select( ) changes the object that displays the currently selected color (object 23 in DEMOPEND).

Another way to do this would be not to depend on the actual parentage of any of the objects in the tree. Being dependent in this manner is not that terrible in light of the fact that DEMO is dealing with user-defined objects. However, there is nothing lost by being direct and naming the actual objects to be changed. The main advantage of the approach used by DEMO is that it provides you with valuable insight into the object system of GEM. By being more direct, however, you make your code more understandable and easier to change.

Now let's talk about the hndl_dial( ) associated with the DEMOPEND tree.

Handling User Input with User-Defined Objects

DEMO enters the hndl_dial( ) routine from pen_select( ) right after the call to set_select( ). We have discussed the main features of hndl_dial( ) in the section dealing with DEMOMENU, but we need to discuss the code inside the forever loop shown in Listing 5.23.

```
FOREVER                   /* Get the user's input */
{
      exitobj = form_do(tree, def) & 0x7FFF;

                          /* Mask out any non-user defined objects.
                             All user defined objects have high byte.*/

      xtype = LWGET(OB_TYPE(exitobj)) & 0xFF00;

                          /* Get out if not user defined */

      if (!xtype) break;

                          /* Go to user defined interpreter */

      xtend_do(tree, exitobj, xtype);
}
```

Listing 5.23: *DEMO User-Defined Input*

Remember that DEMO differentiates between user-defined object selection and GEM object selection through three specially defined object types (0x11B, 0x21B, 0x31B). The only exitobj values that are returned from the form_do( ) are

- The object index of the OK button (object 10)
- The object index of the Cancel button (object 12)
- One of the user-defined object indices.

Note that the user-defined objects must have object flags of TOUCHEXIT, as this is the only way for **form_do( )** to know to return when these objects are selected.

By getting the object type associated with the returned object index, DEMO determines if a user-defined object was selected. (In effect, the masking of the object type will zero out all the predefined object types, thus allowing the break to take effect.) If a user-defined object was selected, DEMO calls xtend_do( ) to process the request, and then because the user may not be finished selecting, reexecutes the **form_do( )**. The only way for the forever loop to stop is for the user to click on the OK or the Cancel button.

User-Defined Command Processing

The routine xtend_do( ), shown in Listing 5.24, takes the comment we made about being dependent on the structure of the object to extremes. It is totally dependent on the structure of the object tree. So be careful if you choose to emulate the style of this code.

```
VOID    xtend_do(tree, obj, xtype)
LONG    tree;
WORD    obj, xtype;
{
        LONG    obspec;

        switch (xtype) {
                case X_SEL:      /* 0x300 - Color selected  */
                        obspec = LLGET(OB_SPEC(obj));

                        obj = get_parent(tree, obj);
                        obj = LWGET(OB_NEXT(obj));

                        LLSET(OB_SPEC(obj), obspec);

                        redraw_do(tree, obj);
                        break;
                case X_FWD:      /* 0x100 - Forward arrow selected */

                        move_do(tree, obj, 1);
                        redraw_do(tree, obj);
                        break;

                case X_BAK:      /* 0x200 - Backwards arrow  */

                        move_do(tree, obj, -1);
                        redraw_do(tree, obj);
                }
}
```

Listing 5.24: *DEMO User Defined Processor*

Since xtend_do( ) only handles the user-defined objects 0x11B, 0x21B, 0x31B, it can be considered the command processor for the user-defined selections. If the user selected one of the visible colors in the selector, then xtend_do( ) must show that selection in the color selection G_BOXCHAR object. xtend_do( ) does this by first obtaining the address of the color pointed at in the object specification field. Then DEMO takes advantage of the object tree structure by getting the parent of the selected object, and uses the NEXT pointer to get the object index of its brother, the G_BOXCHAR color selection object. Given this object index, xtend_do( ) replaces the object specification with the saved object specification of the selected object. The redraw_do( ) only redraws the G_BOXCHAR object.

The other two user-defined selections deal with clicking on the arrows of the color selector. The call to move_do( ) adjusts the items in the selector either one item left or one item right. We won't go into any detail with this code, as it follows the same pattern as set_selection( ).

Highlighting Selected Items

The final topic in our discussion of manipulating objects concerns highlighting selected objects. As we mentioned earlier, these objects are G_IMAGEs, and their object specification fields point to a BITBLK structure that defines the actual image on the screen. DEMO shows that a particular object has been selected by drawing a small box around it through an interesting indirection.

Earlier in the chapter when we discussed demo_init( ), we deferred the explanation of the internals of a function called pict_init( ) because we weren't ready to talk about user-defined objects. We are now ready to analyze pict_init( ), shown in Listing 5.25. pict_init( ) initializes all the G_IMAGE objects in DEMOPEND and DEMOINFD. Actually, all it has to do with the G_IMAGE from DEMOINFD is to transform the format of the BITBLK of the DRI logo to a device-specific format. While it does the same for the images in DEMOPEND, there are additional things that need to be done with these images.

```
VOID pict_init() {

        LONG    tree;
        WORD    tr_obj, nobj;

        rsrc_gaddr(R_TREE, DEMOINFD, &tree);
        trans_gimage(tree, DEMOIMG);

        rsrc_gaddr(R_TREE, DEMOPEND, &tree);
        for (tr_obj = DEMOPFIN; tr_obj <= DEMOEBRD; tr_obj++){
                trans_gimage(tree, tr_obj);

                LWSET(OB_TYPE(tr_obj), G_USERDEF);

                nobj = tr_obj - DEMOPFIN;

                brushub[nobj].ub_code = drawaddr;

                brushub[nobj].ub_parm = LLGET(OB_SPEC(tr_obj));

                LLSET(OB_SPEC(tr_obj), ADDR(&brushub[nobj]));
        }
}
```

Listing 5.25: *DEMO User-Defined Initialization*

All the object handling and user input is handled in the **form_do**( ) in the AES. When the user selects a pencil or eraser form, the AES does not return to the calling program, as it would if the user had selected the OK button or the forward arrow on the color selector. The following mechanism shows how DEMO draws a box around the selected object, while the AES controls the screen (**form_do**( ) is active).

There is an entry point called drawaddr that the AES uses to get back to DEMO. (See the assembler module FARDRAW, Listing 5.26.). The address of drawaddr is placed in the APPLBLK that is pointed at by the object specification of an object in an active dialog. When the user selects the user-defined object, the AES passes control to the address in the APPLBLK, which points to code that draws a box around the selected object. All this is rather sophisticated, which means that a lot of details have to be paid attention to.

```
;/*      FARDRAW.A86      3/22/85       Tim Oren          */
         cseg
         EXTRN   dr_code:near
         dseg
         PUBLIC  drawaddr
         cseg
;        far_draw()
;                ax = hi part of long pointer to PARMBLK
;                bx = lo part of long pointer to PARMBLK
;
;        need to save the regs, get on a local stack, and call
;        the dr_code() routine with the parameters pointed at
;        in PARMBLK
;
;
far_draw:
         push    bp
         mov     bp,sp
         push    ds
         push    es
         push    si
         push    di
         mov     cx,ax              ; remember hi ptr to parm blk
         mov     ax,ss
         mov     drawss,ax
         mov     ax,sp
         mov     drawsp,ax
         cli
```

```
        mov     ax,seg drawstk
        mov     ss,ax
        mov     ds,ax
        mov     es,ax
        mov     sp,offset drawstk
        sti
        push    cx              ; push hi ptr to parmblk
        push    bx              ; push lo ptr to parmblk
        call    dr_code         ; state = dr_code((LONG)pparmblk)
        add     sp,4
        mov     bx,ax           ; remember state
        cli
        mov     ax,drawss
        mov     ss,ax
        mov     ax,drawsp
        mov     sp,ax
        sti
        pop     di
        pop     si
        pop     es
        pop     ds
        mov     ax,bx           ; restore state
        pop     bp
        retf
;
;
        dseg
drawaddr dw     offset far_draw
drawseg dw      seg far_draw
        rw      256
drawstk dw      0
;
        cseg
drawsp  dw      0
drawss  dw      0
        end
```

Listing 5.26: *DEMO FARDRAW Listing*

Remember that we have to do a far call (on the Intel architectures) to get back and forth from the AES using this technique. For this reason, the stack is set up in a particular way. As DEMO does not know how much room is left on the AES stack, drawaddr sets up its own stack to be safe, before calling the code to draw the box. In DEMO, the code that gets executed to handle the drawing is called dr_code( ) and is shown in Listing 5.27. drawaddr provides the connection from the AES, and by declaring dr_code( ) as an external in the FARDRAW code, the AES finally gets a entry into DEMO.

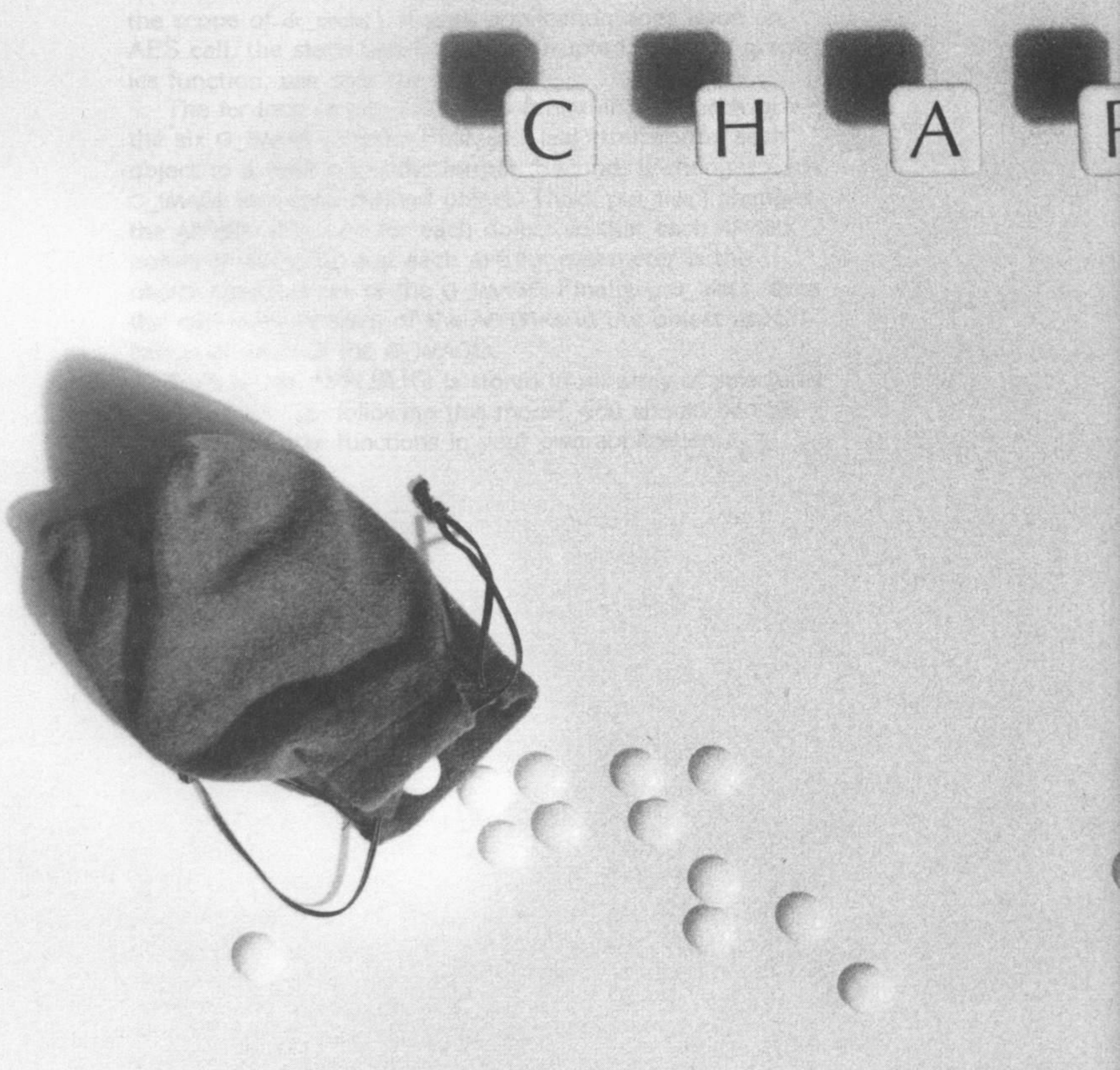
```
WORD dr_code(pparms)            /* Code to handle user defined objects */
LONG    pparms;
{
        PARMBLK         pb;
        WORD            pxy[10], hl, wb;
        LONG            taddr;

        LBCOPY(ADDR(&pb), pparms, sizeof(PARMBLK));


        taddr = pb.pb_parm;
        userbrush_mfdb.mp = LLGET(BI_PDATA(taddr));
        hl = LWGET(BI_HL(taddr));
        wb = LWGET(BI_WB(taddr));
        userbrush_mfdb.fwp = wb << 3;
        userbrush_mfdb.fww = wb >> 1;
        userbrush_mfdb.fh = hl;
        userbrush_mfdb.np = 1;
        userbrush_mfdb.ff = 0;

        pxy[0] = pxy[1] = 0;
        pxy[2] = (wb << 3) - 1;
        pxy[3] = hl - 1;
        pxy[4] = pb.pb_x;
        pxy[5] = pb.pb_y;
        pxy[6] = pxy[4] + pb.pb_w - 1;
        pxy[7] = pxy[5] + pb.pb_h - 1;

        vrt_cpyfm(vdi_handle, 2, pxy, &userbrush_mfdb, &scrn_mfdb,usercolor);

        if((pb.pb_currstate!=pb.pb_prevstate)||(pb.pb_currstate&SELECTED)) {
                if (pb.pb_currstate & SELECTED)
                        vsl_color(vdi_handle,1);
                else
                        vsl_color(vdi_handle,0);
                vsl_width(vdi_handle, 1);
                vsl_type(vdi_handle, FIS_SOLID);

                pxy[0] = --pb.pb_x;      /* Draw a rectangle */
                pxy[1] = --pb.pb_y;
                pxy[2] = pb.pb_x + ++pb.pb_w - 1;
                pxy[3] = pb.pb_y + ++pb.pb_h - 1;
                pxy[4] = pxy[2];
                pxy[5] = pxy[3];
                pxy[3] = pxy[1];
                pxy[6] = pxy[0];
                pxy[7] = pxy[5];
                pxy[8] = pxy[0];
                pxy[9] = pxy[1];
                v_pline(vdi_handle, 5, pxy);
         }
        return (0);
}
```

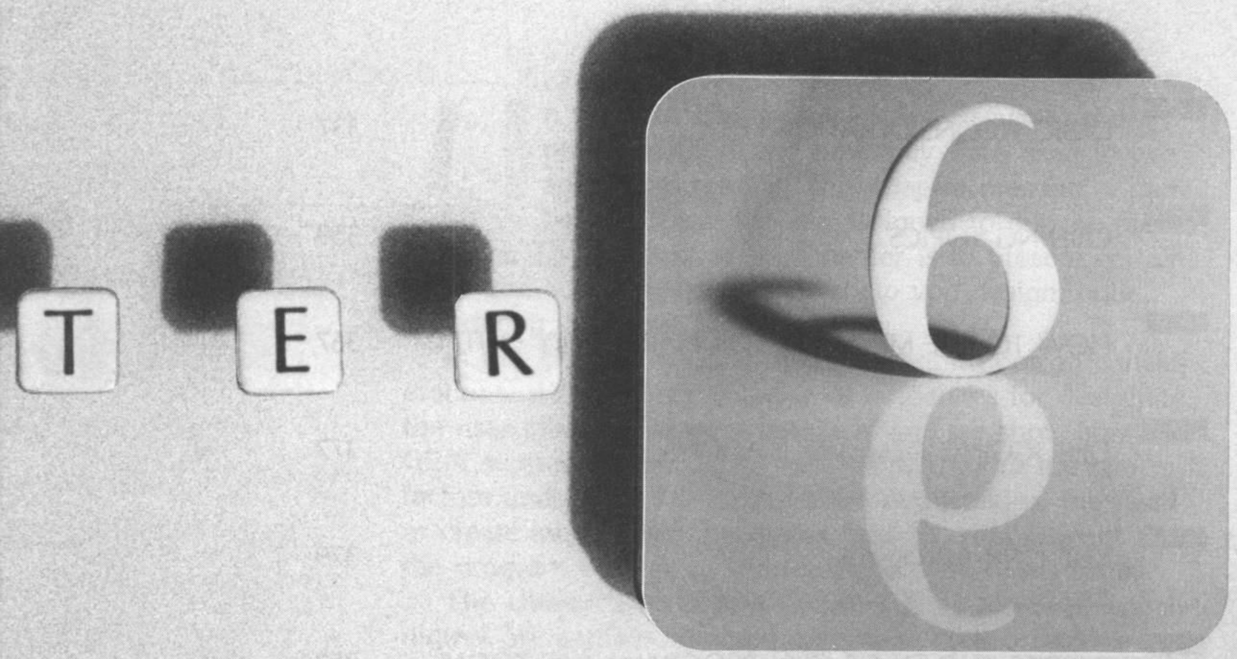
Listing 5.27: *DEMO User-Defined Drawing Code*

This mechanism provides, in effect, a way for you to write extensions to the AES. Because dr_code( ) runs as an extension to the AES, no AES calls can be made within the scope of dr_code( ). If your application does issue an AES call, the stack can become corrupted. For any graphics function, use only the VDI.

The for loop in pict_init( ) does four things to each of the six G_IMAGE objects. First, pict_init( ) transforms each object to a device-specific format. Second, it changes each G_IMAGE to a user-defined object. Third, pict_init( ) changes the APPLBLK structure for each object so that each APPLBLK points at drawaddr, and each APPLBLK parameter is the object specification of the G_IMAGE. Finally, pict_init( ) saves the addresses of each of the APPLBLKs in the object specification of each of the G_IMAGEs.

Each of the APPLBLK's is stored in an array of structures called brushub. By following this model, you should be able to provide similar functions in your own application.


C H A P



ADVANCED GEM TOPICS

In previous chapters, we have discussed the pieces of GEM and how they were used in two small applications. This chapter presents a number of topics that are designed to put the pieces together. The purpose of this chapter is to present certain "nuts and bolts" topics that can help you design, build, and debug your own GEM program.

In this chapter, we first discuss program design considerations. We present a number of guidelines for designing the user interface of your program, and we show how GEM supports these guidelines. We talk about the two factors underlying the typical GEM applications: the need to create event-driven programs, and the requirement that the program be able to redraw the screen at any time.

The chapter also covers certain coding topics and techniques. We explain how you can make your programs more portable to different screen devices and even different microprocessors. We present information about speeding up certain parts of your program, most notably the text display. We also discuss desk accessories.

In addition, the chapter presents examples of how to send graphic output to metafiles (and from there to GEM Draw and OUTPUT.APP). We give you information on debugging your application with GEMSID and tracepoints. We talk about the bindings, and we survey the C compilers available in the different environments.

DESIGN CONSIDERATIONS

The design of a computer program or application is still far from an exact science. Perhaps because good design is so difficult, many programmers prefer to leap into the coding of a program before spending an appropriate amount of time on planning and specifying what the program is to do, or how the program is to accomplish the assigned task. Obviously, there is no substitute for careful and thorough program design. The best programs, and usually the easiest to build and maintain, are well-designed.

Once you know what you are planning to make your program do, GEM provides some implementation aids and offers some clear and consistent philosophy on how your program should be structured. If you follow our guidelines on designing your program's user interface, you can make your program easier to use.

Principles of User Interface Design

The historical antecedents for GEM were the Xerox Star and the Apple Macintosh. The main historical contribution of these products was to make human factors engineering a focus of the microcomputer industry.

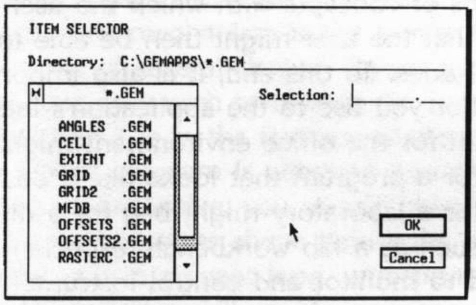
The human factors engineering approach to software design is based on a number of principles. If your program adheres to these principles, it will be easier to learn and use. Here are some of these principles and how they are supported by GEM facilities.

Program with Pictures

One of the guiding principles behind GEM's inception and design is the old adage that "a picture is worth a thousand words." A closely related principle might be stated as "Seeing and pointing is better than remembering and typing." GEM is oriented towards producing high-quality pictures on a variety of computers. For example, GEM supports intensive use of icons. You can use the Icon Editor to create these icons. Your program can display them with either the VDI **vrt_cpyfm**( ) function, or as bit image objects placed in object trees.

Program for External Consistency and Predictability

The next guiding principle you should follow is that your program should behave in a consistent manner. Consistency in a GEM application means external as well as internal consistency. A GEM program is externally consistent if it looks like other GEM applications and is controlled in a similar manner. This is easier when you use features from the AES such as menus and the file selector (**fsel_input( )**). Menu design is covered later in this chapter. The file selector provides a standard method for the user to specify an input or output file (see Figure 6.1). It also allows the user to search anywhere on disk.



```
ITEM SELECTOR

Directory:  C:\GEMAPPS\*.GEM

      *.GEM                Selection:  ________.___

  ANGLES   .GEM
  CELL     .GEM
  EXTENT   .GEM
  GRID     .GEM
  GRID2    .GEM
  MFDB     .GEM
  OFFSETS  .GEM
  OFFSETS2.GEM
  RASTERC  .GEM
```

Figure 6.1: *fsel_input( ) in Action*

Internal consistency is achieved when your program behaves in each situation in a predictable manner. This principle might be called the "principle of least astonishment" because it seeks to eliminate surprises from software. An example of inconsistent behavior would be a control key, such as Escape, that signals an orderly acceptance of some operations and for other operations signals an exit that would abandon the information entered during the operation.

Use Metaphors

Internal consistency might best be achieved by establishing a metaphor for the presentation of your program and adhering strictly to your metaphor. The GEM Desktop is built around the metaphor of a person's desk, with screen icons representing common objects found around the office that are manipulated in a predictable fashion. A folder, for example, can be opened and closed, and can contain documents or other folders. GEM Draw is built on a drafting table metaphor. Draw contains objects such as a ruler and pieces of paper of different sizes. A graphic interface is well suited to creating a variety of metaphors.

The key principle is to present your program's controls in terms of concepts with which the user is already familiar so that the user might then be able to predict what the control does. To this end, it is also important to suit the metaphor you use to the application's intended user. A program for the office environment might be the ideal place for a program that looks like a desktop, but a program for a laboratory might beg for a different kind of desk, such as a lab workbench, especially if the computer is used to monitor and control instrumentation. To support metaphorical programming, GEM provides comprehensive icon support in the VDI, as well as forms and dialogs in the AES.

Keep It Simple

An overly complicated metaphor is difficult to understand. Your program should keep simple operations simple, and ideally it should also make complex operations possible. One method of adhering to this rule is to keep the menu selections relatively brief. Our experience is that seven (plus or minus two) different options is about as many as the average user is able to remember and deal with effectively. Another way to ensure simplicity is to keep the display as uncluttered as possible.

Allow Shortcuts

To speed up the manipulation of your application for your more experienced user, we recommend that you implement keyboard shortcuts for various menu commands. The shortcut for a given menu selection should be displayed to the right of the selection name in the menu. Your program must be prepared to accept menu message events as well as synonymous keyboard events. For further information on design of keyboard shortcuts, see the section called "Other Menu Conventions" later in this chapter.

Provide Feedback

GEM provides many mechanisms for consistent and timely feedback. One mechanism is the mouse form, which you can set to one of several shapes. If your application is accepting text, you may wish to change the mouse form to an I-beam whenever it is in the text acceptance area to indicate this. If your program is about to become compute-bound and thus unresponsive, you should change the mouse form to the hourglass shape before the operation starts. If possible, try to prevent long, unresponsive periods during compute-intensive portions of your program by making regular calls to **evnt_timer**(0) to give other AES processes a chance to share the computer. (This was discussed in Chapter 2.)

Another GEM device for providing appropriate feedback is the form alert mechanism. The alert messages come with one of three different icons (NOTE, WAIT, or STOP) to indicate the severity of the condition.

What You See Is What You Get

Feedback is also provided when you fully utilize GEM's graphics capabilities to display the final result of what the user is trying to produce. Sometimes this is called WYSIWYG, which stands for "what you see is what you get." Many early text processing programs expected complicated text formatting commands (which looked suspiciously like programming commands) interspersed throughout the text file. When you sent the file to a printer, the commands would be processed and the text would be formatted. The problem was that this made it impossible to know certain things about the final product until it was printed. Because, for example, the user couldn't know where the text would cross a page, the process of correctly locating text around page breaks might take several printings. The WYSIWIG feedback rule allows the user to account for things like page breaks early in the document creation process. GEM's graphic text capability allows your program to create the exact image that is created on the output device, whether it be a printer or a film recorder.

Ask for Confirmation

A principle that is similar to feedback is that the application should ask for confirmation. Specifically, your program should allow the user to make reversible decisions quickly, but make irreversible decisions slowly. An example of this principle put into practice can be found in the Desktop. Since DOS does not provide an "undelete" capability, the Desktop has the capability of asking for confirmation whenever the user wishes to delete a file. The Desktop also allows you to disable this capability, which is more friendly for the experienced user.

Another way this principle is applied is when the user is about to exit a program that uses a work area that can be saved to disk. If the user has modified the work area and tries to leave the program, the program may want to check with the user to confirm whether or not the user wants to save the work area. Judicious use of GEM form alerts help keep the user from making a mistake.

Minimize Irreversible Actions

A better approach to handling irreversible decisions or actions is to minimize the actions that cannot be reversed.

GEM Draw has an "Undo" option which allows the operator to take one step backwards and reverse the consequences of the latest change made to the work area.

Prevent Inappropriate Actions

Another way to prevent mistakes as well as teach the user about how to operate your program is to disable menu items when they are not appropriate. GEM provides the **menu_ienable**( ) command to help you do this. When your program has disabled a menu selection, it is printed at a lower brightness level and is not selectable when the user points to it with the mouse. Informing the user before an action rather than after is usually preferable to beeping at the user or displaying an error message when the user has chosen an inappropriate command. Although the user may still wonder why an action is disabled, the way in which you have set up the menu may help them to figure it out for themselves. For example, if they have not yet opened a window by selecting an "Open" item, then you should make sure that the "Close" action is disabled.

Don't Mode Me In

One of the more controversial issues in human factors engineering involves what are called *modes.* A computer has different modes to allow similar keyboard and mouse input to be interpreted differently at different times. A very simple example would be a computer running a calculator program accepting a series of numbers and adding them all together to produce a single answer, and then running a word processor and accepting the same series of numbers and listing them on a page. Another example might be when the user presses Ctrl-C: some programs interpret Ctrl-C as a command to scroll a page (for example), while other programs might expect to be terminated when the user presses Ctrl-C.

Since the computer has only a limited number of inputs, it must use modes to interpret the user's input. The trouble for the user arises when the commands in one mode are inconsistent with the commands in another mode, and when it is not clear to the user at any given time how input will be interpreted by the computer. The proper use of modes is closely related to keeping programs internally consistent.

Use Menu Sidebars

In order to facilitate "modeless" programming, GEM provides menus and windows, and encourages the use of pointing and selection techniques. You can emphasize a certain interpretation of the user's input by tailoring the information displayed in a window. A good example of this can be found in the Draw and Paint applications, which display *menu sidebars* containing icons representing different input states along the left side of the window. The Draw user can then tell when a click-drag draws a line or a rectangle, for example, by which icon has been selected and displayed in reverse video (see Figure 6.2).

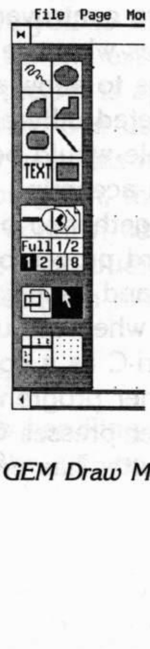


Figure 6.2: *GEM Draw Menu Sidebar*

Consider the Construction Set

Consider using the GEM Resource Construction Set as a model for how you design your program. The Resource Construction Set uses a technique of allowing the sophisticated user to choose from a selection of building blocks at the bottom of the window. This technique makes it very easy to build and organize GEM resources. As the user opens up certain objects to "look inside," the graphical menu changes to reflect a different set of building blocks available. Figure 6.3 shows what some of the RCS building blocks look like. The topmost graphical menu in

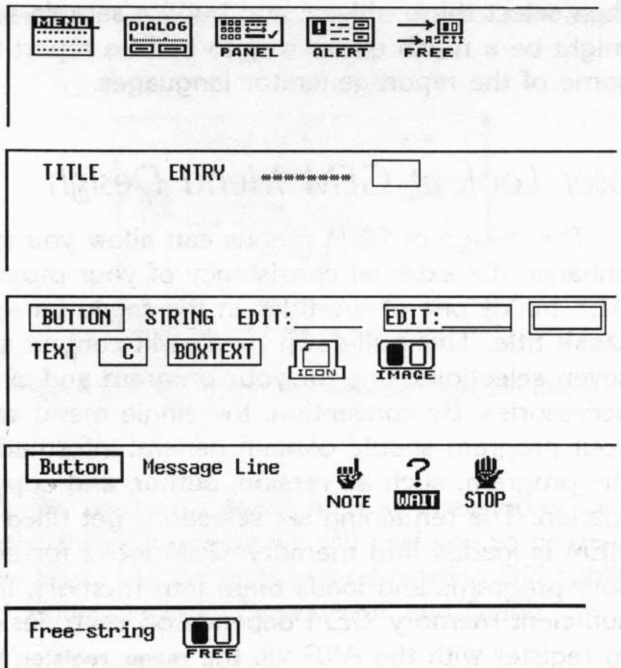


Figure 6.3: *RCS Building Blocks*

the figure contains the five basic building blocks of a GEM resource. When the user selects any of these building blocks, the graphical menu is replaced by a new graphical menu appropriate for constructing the building block. For example, when the menu building block is opened, the graphical menu is changed so that it shows only the menu components as in the second graphical menu in Figure 6.3.

Here is an example of how your program might use a RCS model for generating the format of a report. Your program could display building blocks appropriate for reports (titles, column headers, data fields, sort parameters, and the like) in the menu sidebar. The user could then select these objects and build a sample report. This might be a much easier way to build a report than using some of the report-generator languages.

A Closer Look at GEM Menu Design

The design of GEM menus can allow you to greatly enhance the external consistency of your program. The AES builds one of the titles in the menu bar—namely, the **Desk** title. This pull-down menu will contain at most seven selections: one for your program and six for desk accessories. By convention, the single menu selection for your program should contain general information about the program, such as version, author, and copyright information. The remaining six selections get filled in when GEM is loaded into memory. GEM looks for desk accessory programs and loads them into memory, if there is sufficient memory. GEM depends on each desk accessory to register with the AES via the **menu_register**( ) call.

In older versions of GEM, the **Desk** title was situated in the farthest left portion of the menu bar. When you build a menu with the Resource Construction Set, you can notice that a menu always contains the **Desk** title as the first, leftmost menu title. Starting with version 2 of GEM, this title is automatically moved to the far right of the menu bar by the AES after the program has been loaded, and the name of the program is substituted for "Desk". See Figure 6.4 for an illustration of the **Desk** pull-down menu in version 2 of GEM. Figure 6.5 illustrates the single menu selection specific to the Desktop.

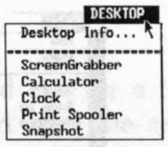
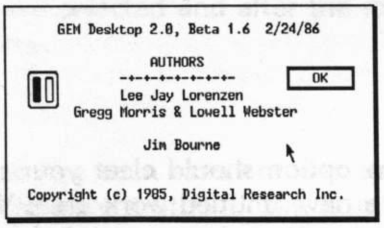


Figure 6.4: *The **Desk** Pull-Down Menu from the GEM Desktop*



```
GEM Desktop 2.0, Beta 1.6  2/24/86

AUTHORS
-+-+-+-+-+-+-
Lee Jay Lorenzen
Gregg Morris & Lowell Webster

Jim Bourne

Copyright (c) 1985, Digital Research Inc.
```

Figure 6.5: *The "About the Desktop" Selection Dialog*

While the remaining menu titles and selections are up to you, in the following pages we will give you some guidelines that have been distilled from several existing DRI applications, including Desktop, Draw, Paint, and Graph.

*The **File** Menu*

Since many applications deal with input and output files, we recommend a **File** pull-down menu as the first (or, on older versions of GEM, the second) title in the menu bar. This menu assumes your program has a work area in memory where the user interacts with your program to do some given task, and that the work area can be read in from and saved out to a disk file. A model of the **File** menu appears as in Figure 6.6.

Let's take a brief look at each selection in the menu.

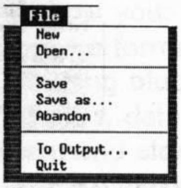


Figure 6.6: *Contents of the File Menu*

*The **New** Option*

The **New** option should clear your program's work area and create a new, untitled work area. You should disable inappropriate menu selections such as **Save** and **Abandon** because there is no current, named work area to save or abandon.

*The **Open . . .** Option*

The **Open . . .** selection should allow the user to open an existing data file and read it into the work area. The three dots after a menu selection are a GEM convention indicating that there is further dialog (or a submenu) should the user select this option. In this case, a file selector dialog is displayed to allow the user to select an appropriate input file.

*The **Save** Option*

The **Save** option should cause your program to save the current work area. If the work area is untitled (that is, if it hasn't been saved before) or if the work area has not been modified since it was read in, this item should not be selectable. Instead, the user should use the **Save As . . .** menu selection, in order to be able to name the data file (and work area).

*The **Save As . . .** Option*

This selection should ask for a file name by using the **fsel_input( )** function, and save the work area to the chosen file name. Additionally, it should name the work area and put the name of the work area in the work area's title bar. As the work area may have been previously untitled, the **Save** and **Abandon** selections should be enabled once this item has been selected and after the user has modified the work area.

*The **Abandon** Option*

If a work area has been saved or was opened from a disk file, it should have a title. In this situation, the user should be able to abandon any modifications made to the work area since the open or last save, and to get a fresh copy of whatever was in the file, by selecting this menu item.

*The **To Output . . .** Option*

Any program that produces information that can be directed to a hard-copy device should have this menu selection. The easiest way for your program to handle this hard-copy requirement is to output a metafile and chain to OUTPUT.APP. We cover these topics in more detail later in this chapter.

*The **Quit** Option*

For the sake of consistency, the user should be able to terminate any GEM program by selecting the **Quit** option in the menu. Your program should check to see if there are any unsaved modifications to the work area and (perhaps optionally) confirm that the user wants to quit without saving.

*The **Preferences** Menu*

We recommend one more menu for the main menu title bar: the **Preferences** menu. This menu allows the user to configure the behavior of the program. For example, your program might want to preselect a certain configuration for menu selections, or it might want to allow the user to turn off the behavior of confirming abandonment of unsaved work areas when **Quit** is selected. We recommend that your program allow these user preferences to be specified in the **Preferences** menu.

We also recommend that you save program configuration information in a known directory such as the directory from which the program was loaded, and that the preferences file has a name of the form *filename*.INF. The command used to load your program (as well as the directory that contained your program) is available via the **shel_read()** function of the AES.

Other Menu Conventions

Most of the rest of the menu options vary too widely between different applications to be standardized. You should, of course, choose menu titles that are consistent with your application's main metaphor.

Another menu convention concerns keyboard shortcuts. We recommend the convention of placing the key sequence of the shortcut to the far right of the selection item. It's also a good idea to have key sequences use the Alt key and to use the convention of combining the small diamond character (ASCII code 7) with the capital letter of the shortcut key. Figure 6.7 shows an example of this.

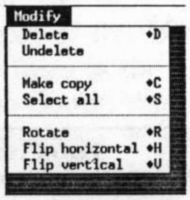


Figure 6.7: *Menu with Shortcuts*

Conventions for the Mouse Form

The mouse cursor form can give the user valuable feedback on what the program is doing. Here are some conventions for mouse form control.

The mouse form should be turned off during draw operations (including those performed by any VDI commands or **objc_draw**( )). The reason for this is that GEM saves a small block around the mouse form and restores this area whenever it moves the mouse or turns off the mouse form. If the program has changed the area around the mouse, it gets overwritten by the contents of the small block when the mouse is next moved.

The mouse form is controlled by calling **graf_mouse**( ), whose prototype we show below. **graf_mouse**( ) takes two parameters. The first parameter, gr_monumber, contains values as defined in Table 6.1. The second parameter, gr_mofaddr, is a pointer to a memory form definition block (MFDB) describing the mouse form, which is used only for user-defined mouse forms. For more information on the mouse form, see the DRI GEM Developer's Kit.

```
VOID graf_mouse( gr_monumber, gr_mofaddr )
WORD gr_monumber;
MFDB *gr_mofaddr;
```

gr_monumber	**Form Name**	**Form Purpose**
0	arrow	General-purpose
1	vertical bar	Text input cursor
2	hourglass	Computer busy signal
3	pointing hand	Selection / Sizing mode
4	flat hand	Selection / Placement mode
5	thin cross hairs	Selection / Drawing mode
6	thick cross hairs	Application-dependent
7	outline cross hairs	Application-dependent
255	user-defined	Application-dependent
256	hide mouse form	Allow drawing operations
257	show mouse form	Restore mouse after drawing

Table 6.1: *Mouse Forms and Their Uses*

In order for the mouse to interact with the Screen Manager and desk accessories in a predictable manner, your application should change the mouse form if it goes outside of your application's window work area. If the mouse form is an hourglass or an arrow, it should be left alone, but any other mouse form should be changed to an arrow. This means that your application must be waiting for a mouse rectangle event (using **evnt_multi()**), with the rectangle set to the window work area. When the mouse leaves the work area, your program should call **graf-_mouse()** to set the form to an arrow. When the mouse comes back into the work area, your application should set the mouse form back to the appropriate cursor form for your application. See the GEM DEMO program in Chapter 5 for an example of this behavior.

The primary mouse form is the arrow, which has gr_monumber 0. When in doubt, this is the form to use.

The hourglass (gr_monumber 2) should be used whenever your application is performing a time-consuming operation, such as a file open, read, or write. This is to make sure the user is informed that the application is very busy.

Other mouse forms, their gr_monumbers, and their intended uses can be found in Table 6.1.

Conventions for Mouse Techniques

There are a number of well-established conventions for keyboard usage on computers, such as using the Shift and Ctrl keys to change the meaning of other keys. Other conventions are less firmly established, such as interrupting a program by pressing Ctrl-C. Since the mouse is a relatively new input device, we have included information here about what kind of behavior with the mouse has been given meaning by various applications.

The mouse is used as a pointing device, and GEM takes care of the correspondence between mouse and cursor movement. We assume that there is only one button on the mouse because we can be fairly certain that there will always be at least one button, and also because we feel that it is easier for the user to use if there are a minimal number of controls. We will therefore only talk about one-button mouse techniques.

The single-click technique involves pressing the mouse button once and then letting up. We suggest that your program handle this as the user's way of indicating selection of an object or control, and that your program cause whatever is under the mouse to be displayed as selected. Once an object has become selcted, certain actions in the program's menu may become appropriate (or not). As discussed in the section called "Prevent Inappropriate Actions", your program should enable or disable the various menu items which are appropriate to use with the selected object. Use the AES function menu-ienable( ) to do this.

The double-click technique requires the user to press the button twice in a row, very quickly. The **evnt_multi( )** function is configured to wait for a certain amount of time once it detects a single-click, in order to see if the user wants to double-click. The user can set this waiting period from the Desktop, or your program can set it with the **evnt_dclick( )** call: longer periods make it easier to do the double-click, but they impede the more sophisticated user of your program. The double-click is usually taken to indicate the equivalent of an Open menu request on the object under the cursor.

Some programs may attach meaning to more than two clicks, but we have not found it easy to triple-click with very great accuracy.

Another technique is called a *click-drag,* which refers to the action of positioning the mouse cursor, pressing the button, moving the mouse cursor, and releasing the button. This technique is often used for several different actions, depending on what the mouse cursor is positioned on as well as on the state of the program. Possible meanings of the click-drag include

— Select an object and move or copy it.
— Select a group of objects.
— Draw a shape, or track and draw at the cursor's position.

There are many variations on the click-drag technique, most of which involve pressing a key on the keyboard (like Shift or Ctrl) before pressing the mouse button. The Resource Construction Set and Draw, to name two examples, use these techniques to aid power-users. These techniques are harder to document and remember, and so they should be used sparingly.

Some Ideas on How To Indicate Selection

This section presents some ideas for indicating object selection. We use the term "object" loosely here, as we mean what appears as an object to the user (which may or may not be an AES object). There are many different ways of indicating object selection, partly because what you can do with selected objects varies from application to application. Thus, what we will present here constitutes ideas rather than conventions.

An AES object has a "selected" attribute, and when the AES draws a selected object, it inverts the object. For bit images and simple rectangles, an object can be inverted by calling **vro_cpyfm**( ) with the XOR logic operation (see Chapter 3). An example of this can be found in the Desktop, which uses this technique extensively to indicate selected file icons.

Another method of indicating selection is to enclose the object with a rectangle (drawn in XOR mode, so that it can easily be removed). The rectangle may have a number of control points which are represented as very small filled rectangles located in strategic parts of the rectangle. For example, placing the control points in the corners of the rectangle indicates a movable object. Placing control points in the midpoints of the rectangle segments is a good way to indicate scaling points that allow the object to grow or contract.

GEM Graph draws control points around the shape of the object (instead of enclosing the object in a rectangle) to define the selected object as precisely as possible.

Still another method of indicating object selection might be to change the mouse cursor form whenever the mouse moves over the top of the selected object. This would be difficult to do with multiple object selection in GEM, because **evnt_multi**( ) can only wait on two mouse rectangle events. Draw uses a related technique to indicate whether the point the user has selected on the object moves the object (Draw uses the flat hand mouse form) or scales it (Draw uses the finger pointer).

GEM Program Structure

In the two sample applications in Chapters 4 and 5, we have already introduced the structure that is implied by an event-driven program. We are distilling this program structure again here to emphasize the effect it has on the design of your GEM program.

The most noticeable detail about the structure of our examples is that they consist of the following components:

```
initialization( );
while( not_done )
{
    evnt_multi( . . . );
    handle_the_events( );
}
termination( );
```

Much of the input for a GEM program is received in one place: namely, the **evnt_multi()** call. This leads to a relatively clean, flat design that minimizes the number of different modes your program contains.

You may want to pay particular attention to where your program design deviates from this model. You may notice that calls to **form_do()** and **form_alert()** are obvious deviations from the modeless model. These are clear examples of program modes, where the user is required to respond immediately to the dialog or alert, and the input is interpreted in a manner unique to each form. Another example of modal input comes in the DEMO program when the program is actually drawing on the screen. These are necessary parts of the program, but we highly recommend that these modal parts of your program be as isolated and as cleanly defined as you can make them. In other words, keep the modal behavior in your program focused towards a single purpose.

One other requirement of GEM markedly affects program design. Since GEM is multitasking and since the user may open a number of windows besides the window(s) of your application, your program may be required to redraw its display at any time. The flat, modeless program structure we demonstrated above is ideally suited to handling arbitrary redraw messages.

This differs from a simpler program output model in which once your program outputs something to the screen, you expect it to stay put. Your GEM program must be prepared to rebuild the sand castle any time a wave has washed it away. You should use the **wind_update** (BEG_UPDATE) call around the handle_the_events() code to prevent any waves from coming in while the program is building or changing the sand castle. In order to share the beach, however, the program must release the update region after it has finished by calling **wind_update** (END_UPDATE).

To put this more directly, the program structure we recommend isolates all of the dedicated modal behavior inside the handle_the_events( ) code and thus inside the wind_update( ) protected area. This is another reason to keep the program modes focused and single-purposed, as the time the program spends in the protected region is a time when other applications cannot display information. Thus, you may want to use modeless windows for time-consuming input operations (for example, extended text input) instead of modal dialogs, so that the user can access other parts of GEM (like the desk accessories).

There are a number of possibilities for how your program can handle the redraw requirement. Perhaps the most simplistic method is illustrated by the DEMO application. DEMO draws to the screen and then saves the entire screen area after each modification of the screen area (using a BITBLT VDI operation, **vro_cpyfm( )**). This provides for a simple redraw model, at the cost of large areas of memory, for the screen save area.

Another method is the extensive use of object trees. Object trees are ideal for holding rectangularly oriented text displays (like forms), and they have the hooks to handle bit images (icons) and user-defined objects. The user-defined objects are like wildcards, in which a procedure in your program is called whenever the object is to be drawn. The procedure is called with a pointer to a data structure containing the screen location where your program should draw the object. A big advantage of managing your display with object trees is that the Resource Construction Set can help you build a prototype of the display before you even start to write any code. Another advantage of objects is that they make it easier for your program to determine what the user has selected because the **objc_find( )** function accepts the current mouse coordinates and returns a pointer to the object at those coordinates. The drawback of object tree display is the limited variety of drawing objects, the relatively slow pace of object drawing, and the difficulty you may have in making your program output the contents of the object tree to a metafile.

Programs like Draw manage their own display data structures and make extensive use of the VDI for their display. Draw collects lists of display items, and it performs all selection and redraw tasks based on its own data structures. Your application can do this, too, especially if it has requirements for more sophisticated graphics output than the object trees can reasonably support.

In general, we recommend that you use objects (and user-defined objects) to display simple items like boxes and text as well as selectable items such as icons. We suggest that you use regular VDI commands for other kinds of displays, such as charts and graphs.

CODING TOPICS

In this portion of the chapter, we will discuss a number of topics that may affect how you code your program. Once you've decided what you want your program to do and how you want your program to do it, you still have to make many smaller decisions, such as what colors to use or how to have your program produce printed output. In the pages that follow, we will present information that will aid you in making those decisions.

You may or may not be interested in having your program run on more than one machine, or in having your program produce high quality output on many kinds of devices. In this section we discuss how you can write programs to be portable, because we have found that writing portable programs from the start is only slightly more difficult than writing nonportable programs, but that porting a nonportable program once it has been written is much harder. Furthermore, the microprocessor industry is still changing quickly enough to justify a little extra effort, just in case your computer supplier goes out of business, or in case somebody builds a computer that works much better than the one you've got now. The odds are that GEM will work on it—why not your program, too?

We will also talk about how you can make your program run faster, often with only a little more effort if you know how from the start. We will discuss desk accessories because they are only slightly different than normal GEM programs.

Making Graphics Programs Device-Portable

One of the most important design goals of GEM has been to make it possible to write graphics-based programs that are portable to many different machines. This portability is possible to achieve, but it can be a bit tricky. Here are some of the things that you have to be careful about.

First, we recommend that your program use its own world coordinate system and that it translate its coordinates into raster coordinates when displaying to the screen device. Although this is more difficult than using straight raster coordinates, it is especially important when your program wants to output the image to a metafile, since metafiles can accept your world coordinates directly (with **vm_coords**( )), and since you can achieve higher quality output when you map higher resolution world coordinates onto different devices that have lower (and varying) degrees of resolution.

Since aspect ratio differs from device to device, you also need to account for it. Thus, you will probably want to use generalized drawing primitives like **v_ellipse**( ) instead of **v_circle**( ) so that you can control aspect ratio from your program. It is especially important to use world coordinates and account for aspect ratio in order to maintain relationships between graphical objects on the screen. For example, if you use **v_circle()** and don't adjust for aspect ratio, a line drawn tangent to a circle on one device may not be tangent to the circle when you draw these objects on a device with a different aspect ratio.

GEM makes it easy to write programs that have colorful graphics, but not all devices have the same colors. By planning carefully, you can write programs that work on both color and black-and-white systems. If you use color to differentiate information, you may also plan on how you might communicate the same information in black-and-white. GEM offers a variety of pattern fill functions that work well in black-and-white environments.

The raster operations require a certain amount of care. If you closely examine different screen devices, you may be surprised to find out how many kinds of different methods there are to do raster graphics. Without GEM, it would be extremely difficult to write programs that perform fast raster operations on more than a single type of machine. With GEM, your program can be made to run on very different machines, if you watch out for some details. One of these is screen resolution, since an icon may look very different on a low-resolution screen than on a high-resolution screen. (The Desktop has two sets of icons—one for low resolution screens and another set for high-resolution screens.) Another detail is programming in color environments, where you must be careful to allocate enough memory for more than one bit plane. A third detail is knowing about the standard format that GEM uses to hide device representation details covered in the section on raster operations in Chapter 3.

Although GEM supports many different devices, the capabilities of these different devices causes GEM to offer different levels of support for different devices. For example, raster operations are only supported on screen devices. GEM also allows you to direct graphics operations to metafiles, which programs like GEM Draw can accept as input. Metafiles can also be taken by the OUTPUT application and displayed on screen, printer, plotter, or film recorder. We will discuss metafile output later in this chapter.

Making GEM Programs Processor Portable

There are a number of significant differences between the Intel 8086/88 microprocessor found in the IBM PC and compatibles and the Motorola 68000 found in the Atari ST. Nevertheless, there are programming conventions and techniques you can use to make it relatively easy to port your program between these two different kinds of machines.

First, let's talk about the portability conventions file, PORTAB.H. This header file contains a number of synonyms for common C data types, including

WORD	A signed short integer (16-bit) quantity.
LONG	A signed 32-bit integer.
NULLPTR	0 cast to a pointer.

The main reason for these synonyms is that different compilers may implement integers in different lengths. For example, some C compilers on the 68000 implement type int as a 16-bit quantity, while others use 32 bits. PORTAB.H provides synonyms that you can redefine to whatever the compiler requires to make the data type truly (in the example of WORD) a signed, 16-bit value.

The LONG values are frequently used as pointer values in the AES. The reason for this is to allow your program to be compiled in the most efficient memory model for the 8086 (called the *small* model because all of the pointers are 16 bits) and still have access to resources, which may be loaded outside of the 64K area addressable by 16-bit pointers. On the 68000, all pointers are treated as 32-bit values, which allows very easy access to the resource values. The LONG pointers correspond to the mixed memory model constructs (most notably, the far pointer) found in a few of the 8086 C compilers.

The NULLPTR is provided for use in place of NULL or 0 for situations in which the integer size is 16 bits, but the pointer size is 32 bits. Consider what happens when passing in a 16-bit NULL value when the procedure is expecting a 32-bit pointer; your C compiler simply extracts the 16-bit NULL plus whatever 16-bit value follows the NULL on the stack, and would construct a value which was not NULL and not a valid pointer. Always use NULLPTR instead of NULL or 0 when you want to pass or test for a pointer type with value 0.

Another useful header file is MACHINE.H which contains processor-dependent declarations for routines such as ADDR( ). On the 8086, ADDR( ) is a LONG-valued function that takes a 16-bit pointer (in small model) and adds the segment value to return a full 32-bit pointer. On the 68000, ADDR( ) is defined as the original pointer, which is already 32 bits. Several other functions are declared that allow you to copy, set, or access values that may be anywhere in memory on an 8086. All of these routines are greatly simplified with the flat addressing space of the 68000.

Another useful fact to remember when porting programs is that the byte ordering is different on the two machines. Consider the following fragment of C code:

```
WORD ival;
BYTE *pival;
. . .
ival = 1;
pival = (char *) &ival;
```

On the 8086, *pival points to a byte with a value of 1; on the 68000, *pival points to a byte with a value of 0. This is because the 8086 always stores the low-order byte first in memory, while the 68000 always stores the high-order byte first. MACHINE.H contains a number of macros that allow you to pack and unpack byte values into WORD and LONG values in a manner independent of the processor.

One of the problems that we encountered in the development of our VDI examples in Chapter 3 was when we tried to create bit images using LONG values in CPYTRAN1.C and CPYTRAN2.C. This technique worked on the 8086 GEM. Since raster images are WORD-aligned in standard format, however, the words were swapped (relative to the 8086), and these programs failed on the 68000. Always use WORD values to initialize bit images.

Atari Hints

We found a difference in GEM on the Atari ST that can be very puzzling. The older versions of Atari GEM do not include the GDOS, which is a part of the VDI that is designed to make the VDI more device-independent. Because this was left out, older Atari computers cannot run any of our VDI examples that used NDC space or any examples that loaded fonts with the **vst_load_fonts( )** call. The way to know if the GDOS is loaded correctly is to watch for the words "GEMVDI resident" when you are booting your system. The GDOS load file we used was called "ATARIGD.PRG" in the \auto directory on the boot disk. As of this writing, Atari has indicated that they will ship the GDOS, and we assume that they will use the conventions we have described.

We found a couple of other inconsistencies in the Atari port. First, **fsel_input**( ) automatically converts a file name to uppercase on the 8086 version of GEM that we used, but not on the Atari. In fact, the Atari version will only accept uppercase letters and will not display any lowercase letters in the **fsel_input**( ) call! Second, **vro_cpyfm**( ) ignores the destination rectangle size on the 8086 GEM we used, but the destination rectangle had to be the same size as the source rectangle on the Atari.

Making GEM Programs Faster

Because many GEM functions need to display textual information, the VDI has been extensively optimized in order to be able to display text in the system font as quickly as possible. The VDI allows you to display text with a large degree of control over the detailed appearance of the text. If you just want to display text as quickly as possible and you're not concerned about varying the font or type size, here are some things you can do to take advantage of the extensive text optimizations.

The first and most important improvement you can make to your program is to use the optimized system font and to display text on byte boundaries (or, better yet, word boundaries). What this means is that you should make certain that the raster address of the starting pixel in the horizontal (x) direction is evenly divisible by 8 (or 16, for word alignment). This alignment facilitates transferring the font information because the screen driver doesn't need to do as many shifting and logic operations.

The second improvement you can make is to be certain that your text does not get clipped. The screen driver checks to see if it needs to clip part of the text of your program. If it does need to clip your program's text, it must handle the boundary conditions very carefully to transfer portions of the clipped characters. If, however, your program checks the length of the string and only displays the number of characters that are fully displayable within the clipping area on the screen, the text display is noticeably quicker.

Making GEM Desk Accessories

GEM desk accessories are similar to normal GEM applications with a number of exceptions. These programs are identified by having a file extension .ACC, and they must be placed in the \gemboot directory in the 8086 environment and in the \auto directory on the Atari. The programs are loaded when GEM is loaded, and if there is not enough memory, they do not get loaded.

To allow the user to run a desk accessory, the desk accessory program must use the **menu_register**( ) call to inform the AES that it is loaded and ready to be called. The application program must also display a menu bar in order to give the user a means of invoking the desk accessories from within the application. The VDI example programs we presented in Chapter 3 (CTRLNDC.C and CTRLRC.C) do not show a menu bar, and thus desk accessories cannot be used when these programs are running. We present another driver program here in Chapter 6, called CTRLMETA.C, that allows you to run desk accessories while the VDI example programs are running.

Only three total desk accessory programs can be loaded in current versions of GEM. Each desk accessory can make one or more calls to **menu_register**( ), however. This means that one desk accessory could conceivably use all six slots in the Desk menu.

In versions of GEM prior to 1.2 (including the Atari version, as of this writing), desk accessories could not use menus. Thus, your desk accessory needed to be controlled either through the keyboard or through the user's selection of icons in the desk accessory window, as does the GEM Desk Calculator.

A desk accessory starts up like a normal application, except that it calls **menu_register()** and **evnt_multi()** before opening a window. Upon receipt of an AC_OPEN message, your application should open the window and do its thing. Upon receipt of the AC_CLOSE message, your desk accessory program must assume that the user has finished running an application, and that the AES is also wiping out all of the windows (including your program's), which means that your desk accessory must create and open a window again after every AC_CLOSE message.

The best way to debug a desk accessory is to first develop it as a normal GEM application. This allows you to use either of DRI's symbolic debuggers, SID or GEMSID. Then, when your program is working, insert the extra statements (that is, **menu_register()** and the code to handle AC_OPEN and AC_CLOSE), rename the file, and place it in the appropriate boot directory. The sample program in Chapter 4, HELLO, is included as a desk accessory in the DRI Developer's Kit with conditional compile indicators (which we removed to simplify our example) around the code necessary to turn it into a desk accessory. See the DRI Developer's Kit for more information.

HOW TO USE METAFILES FOR HARD-COPY OUTPUT

Metafiles are the best way in GEM to get hard-copy output from your program. Although you can open the printer device, for example, directly from your program, we do not recommend this. Instead, if you produce a metafile, you can pass this file to OUTPUT.APP, which can then display it on a printer, a plotter, a film-recorder, or as a part of a slide show on the screen. We explore here a few of the issues involved in drawing your output to metafiles.

A *metafile* is a recording mechanism (that is, a disk file) that allows graphical images to be specified in a device independent fashion. The primary purpose of metafiles is to contain graphical information that can be transferred between different devices or applications. In this way, metafiles are to GEM what text files are to word processing programs.

The main reason that metafiles are difficult to use is that you must have a clear understanding of the different coordinate systems involved. You must also realize that aspect ratios are different in other devices, and thus you must use the VDI calls with a lot of attention to portability if you want your images to look similar from device to device.

A metafile consists of a page size, a coordinate system, and a series of graphics commands. A metafile workstation is opened in a manner very similar to opening a screen workstation, with the device id (in workin[0]) specifying that this device is a metafile (see Table 3.2 for device ids). You should then set up the page size and coordinate systems for the metafile, and draw your figure much as you would to the screen. Finally, make sure you close the metafile workstation so that the file gets closed properly.

We will discuss the following functions in this section:

vm_filename( )	Change GEM VDI File Name
vm_pagesize( )	Define Physical Page Size for Metafile
vm_coords( )	Define Coordinate System for Metafile

Changing the GEM VDI File Name: **vm_filename()**

The standard destination of a metafile is to a file called GEMFILE.GEM in the default directory. Use the **vm_filename()** function immediately after you open the metafile workstation to direct metafile output to the file you specify in the null-terminated file_name variable. The prototype for the **vm_filename()** function looks like this:

```
void vm_filename( m_handle, file_name )
WORD m_handle;
BYTE *file_name;
```

For an example of **vm_filename()** in use, see the open_meta( ) function in Listing 6.1 later in this chapter.

Note that **vm_filename()** closes the current metafile and reopens the file you specify. Thus, any output to the metafile before this call is lost in the previous metafile (GEMFILE.GEM).

Defining a Coordinate System for a Metafile: **vm_coords()**

One of the nicest aspects of drawing to metafile devices is that you can specify an arbitrary world coordinate system. Thus, most applications use the world coordinates that are most convenient for the application to work in, which means that applications don't have to specify arbitrary raster coordinates. If you don't specify a coordinate system in a metafile, NDC space is assumed.

To specify metafile coordinates, give the x and y positions of the lower left and upper right corners of your image. Most VDI functions behave the same whether you give lower left and upper right diagonal endpoints, or upper left and lower right diagonal endpoints. The **vm_coords()**

function actually causes OUTPUT.APP to rotate your image to make the first point you specify appear in the lower left corner of the final image.

Call the **vm_coords()** function as follows:

```
void vm_coords( m_handle, low_left_x, low_left_y, hi_right_x,
                hi_right_y )
WORD m_handle;
WORD low_left_x, low_left_y;
WORD hi_right_x, hi_right_y;
```

Here are a couple of examples. Both of these assume that your image uses NDC space as your world coordinates. In the first case, suppose you make the following call before writing the rest of the image to the metafile:

```
vm_coords( m_handle, -32768, -32768, 32767, 32767 );
```

When OUTPUT.APP displays the metafile, all of the NDC coordinates of your image are mapped into the upper right quarter of the page.

In our second example, suppose you use this call:

```
vm_coords( m_handle, 0, 32767, 32767, 0 );
```

OUTPUT.APP displays the metafile so that the 0,0 point is in the upper left corner. This orientation corresponds to the RC method.

Defining Physical Page Size for a Metafile: **vm_pagesize()**

The **vm_pagesize()** function stores information in the metafile to indicate that the image is intended to be reproduced on a page of the indicated size. Call this function as follows:

```
void vm_pagesize( m_handle, width, height )
WORD m_handle;
WORD width, height;
```

The setting of the page size can be a bit of a challenge. The units of measure used for setting the page size are tenths of millimeters (.1 mm). This unit was chosen because there are 254 tenths of millimeters per inch, and thus conversion between metric and English measures is fairly easy. The OUTPUT application of GEM is fairly versatile and partitions the image into page-sized pieces if the image is too large for a single page and the user hasn't specified the "Best Fit" option of OUTPUT. The GEM Draw application uses several different page sizes, starting from 7½ inches by 10 inches.

The page size and coordinate system you use determine the aspect ratio of the image. Thus, if you're using NDC coordinates as your world coordinates, you can specify a square image with an aspect ratio of 1 (equal height and width). This leaves you with an image, however, that does not fill your typical 8½ by 11 inch piece of paper.

HOW TO SET THE METAFILE WORLD COORDINATES CORRECTLY

When setting your metafile world coordinates, make sure that the numerical value of your coordinate scaling is greater than the numerical value of the page size (in tenths of millimeters). This gives the OUTPUT application greater accuracy in its scaling. Furthermore, older versions of OUTPUT.APP crash if you ignore this restriction. This is most likely to happen if you try to map the raster coordinates of your screen directly onto a metafile.

Unfortunately, the GEM designers didn't include the functions **vm_pagesize()** and **vm_coords()** in the VDI bindings in the Developer's Kit. You will, however, find the bindings in source form in Listing 6.1.

Some Guidelines for Using Metafiles

Because metafiles can be directed to any device supported by GEM, you need to be careful about which VDI functions you use with metafiles. Here is an account of some of the things to be careful about when performing output to a metafile:

First, don't depend on all return values from functions such as **v_opnwk()** or **vst_height()**, since metafiles don't really have discrete line thicknesses, polymarker heights, or character heights. Use functions like **vst_point()**, which sets the character height in absolute terms (that is, in points, where 1 point equals 1/72 inch).

Second, try to use GDP functions, which are supported on all devices. Specifically, the fill value in **vr_recfl()** is ignored in a metafile, whereas **v_bar()** can probably produce an equivalent effect.

Third, you should note that while the AES has a number of useful output functions, including **objc_draw()** (which is covered in Chapter 2), there is no way in the current version of GEM to output AES information into a metafile. This means that to output any objects that you normally draw with **objc_draw()** to a metafile, you will have to write your own routine to walk the AES data structures and output the items individually to your metafile.

Fourth, it's important to note that raster operations are totally unsupported in metafiles. See Appendix B for a list of VDI functions supported by the metafile driver.

Finally, although GEM Draw will accept metafiles and allow the user to change them, there are some VDI functions that are supported in metafiles but that are not supported by Draw. Specifically, Draw does not support clipping (**vs_clip( )**) and different writing modes (**vswr_mode( )**).

An Example of a Metafile Output Routine

In Listing 6.1, we present a program called CTRL-META.C that is a replacement for the CTRLRC.C module presented in Chapter 3. The purpose of the program is to set up an environment for calling draw_rc( ), with a few changes. The major change is to add a menu bar and to make the routine draw within a window. We also have used an **evnt_multi( )** call to allow the user to terminate the program with either a mouse button or a keyboard event.

```
/* CTRLMETA.C - routine to call draw_rc() routine (using Raster Coords);
       - provide a simple menu to enable access to Snapshot Accessory
       - provides meta file output from menu */

#include "portab.h"
#include "machine.h"

WORD contrl[12], intin[128], ptsin[128], intout[128], ptsout[128];

#define SOLID 1
#define WHITE 0
#define SQUARED 0
#define BEG_UPDATE 1
#define END_UPDATE 0
#define WF_TOP 10
#define R_TREE 0
#define OBJSIZE 24

/*   "ctrlmeta.h" - built by RCS     */
#define CTRLMENU 0        /* TREE */
#define TITLEOBJ 3        /* OBJECT in TREE #0 */
#define MFILEOBJ 6        /* OBJECT in TREE #0 */

/* write the page size to the metafile header */
    VOID
vm_pagesize( handle, pgwidth, pgheight )
WORD handle, pgwidth, pgheight;                   /* in .1 millimeters */
{
       contrl[0] = 5;                      /* opcode */
       contrl[1] = 0;                      /* # input vertices */
       contrl[3] = 3;                      /* len intin array */
       contrl[5] = 99;                     /* function id */
       contrl[6] = handle;
```

```
        intin[0] = 0;                   /* sub opcode */
        intin[1] = pgwidth;             /* width in .1 mm */
        intin[2] = pgheight;            /* len in .1 mm */

        vdi();
}

/* write the world coordinate system to the metafile header */
    VOID
vm_coords( handle, llx, lly, urx, ury )
WORD handle, llx, lly, urx, ury;
{
        contrl[0] = 5;                  /* opcode */
        contrl[1] = 0;                  /* # input vertices */
        contrl[3] = 5;                  /* len intin array */
        contrl[5] = 99;                 /* function id */
        contrl[6] = handle;
        intin[0] = 1;                   /* sub opcode */
        intin[1] = llx;                 /* lower left x coord */
        intin[2] = lly;                 /* lower left y coord */
        intin[3] = urx;                 /* upper right x coord */
        intin[4] = ury;                 /* upper right y coord */

        vdi();
}


/* open a metafile and change its name to whatever the menu's name is */
VOID open_meta( work_in, pm_handle, work_out )
WORD *work_in, *pm_handle, *work_out;
{
    BYTE fname[20];
    LONG mobj, nmptr;
    WORD fnlen;

    rsrc_gaddr( R_TREE, CTRLMENU, &mobj ); /* grab 1st tree (menu) */
    mobj += TITLEOBJ * OBJSIZE + 12;     /* go grab OB_SPEC of 4th object */
    LBCOPY( ADDR(&nmptr), mobj, 4 );     /* grab the 4 bytes of OBSPEC */
    fnlen = LSTRLEN(nmptr);              /* find its length */
    LBCOPY( ADDR(fname), nmptr, fnlen );/* copy the name to local area */
    while( fname[--fnlen] == ' ' ) ;     /* eliminate trailing blanks */
    fname[++fnlen] = '.';                /* append extension ".GEM" */
    fname[++fnlen] = 'G';
    fname[++fnlen] = 'E';
    fname[++fnlen] = 'M';
    fname[++fnlen] = 0;
    for( fnlen=0; fname[fnlen] == ' '; ++fnlen )
        ;                                /* eliminate leading blanks */
    v_opnwk( work_in, pm_handle, work_out );
    vm_filename( *pm_handle, fname+fnlen );/* go change its name */
}


/* set up for metafile output and call draw_rc routine */
VOID meta_out( width, height )
WORD width, height;
{
    WORD m_handle, work_in[11], work_out[57];
    WORD ii, junk;
    WORD maxw_m, maxh_m, swidth_tmm, sheight_tmm;
    WORD pxy[10];

    for( ii=0; ii<11; ++ii ) work_in[ii]=1;/* init work_in array */
    work_in[10] = 2;                      /* use RC coordinates */
    work_in[0] = 31;                      /* Metafile device number */
    open_meta( work_in, &m_handle, work_out );/* open the metafile */

    maxw_m = 10*width; maxh_m = 10*height;
    swidth_tmm = (254*15)/2;              /* 25.4 mm/inch, 7.5 inches */
    sheight_tmm = (254*10)/2;             /* 5.0 == 10/2 */
    vm_pagesize( m_handle, swidth_tmm, sheight_tmm );
    vm_coords( m_handle, 0, maxh_m, maxw_m, 0 );

    pxy[0] = pxy[1] = 0;
    pxy[2] = maxw_m; pxy[3] = maxh_m;
    vs_clip( m_handle, 1, pxy );          /* safety precaution: do clip */
    vst_point( m_handle, 14, &junk, &junk, &junk, &junk );

    draw_rc( m_handle,0,0,maxw_m,maxh_m );/* do it to metafile */

    pxy[0] = pxy[1] = pxy[8] = pxy[9] = 0;
    pxy[2] = maxw_m; pxy[3] = 0;
    pxy[4] = maxw_m; pxy[5] = maxh_m;
    pxy[6] = 0; pxy[7] = maxh_m;
    vsl_ends( m_handle, SQUARED, SQUARED );/* reset to default */
    v_pline( m_handle, 5, pxy );          /* delimit screen with a box */

    vs_clip( m_handle, 0, pxy );          /* safety precaution: end clip*/
    v_clswk( m_handle );                  /* flush it and close */
}

/*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\*/

VOID GEMAIN()
{
    WORD handle, work_in[11], work_out[57], w_handle;
    WORD pos_x, pos_y, max_w, max_h, pxy[10];
    WORD ii, ev_which, junk;
    LONG mbar, pevbuff, mobj;
    WORD evbuff[8];

    appl_init();                                   /* init AES for next call */
    if( ! rsrc_load(ADDR( "CTRLMETA.RSC" )) ) { /* built with RCS */
        form_alert( 1, ADDR("[3][Unable to load|CTRLMETA.RSC][Abort]"));
        return;
    }
    rsrc_gaddr( 0, 0, &mbar );                     /* Tree 0 is menu */
    pevbuff = ADDR( evbuff );                      /* need this later */

    handle = graf_handle( &ii,&ii,&ii,&ii );       /* get screen handle */
    v_clrwk( handle );                             /* clear workstation */

    for( ii=0; ii<11; ++ii ) work_in[ii]=1;        /* init work_in array */
    work_in[10] = 2;                               /* use RC coordinates */
    v_opnvwk( work_in, &handle, work_out );        /* open the workstation */

    menu_bar(mbar,1);                              /* display BEFORE wind_open*/
    wind_get( 0,4,&pos_x,&pos_y,&max_w,&max_h );/* find Desktop work area */
    max_w -= 1;                                    /* leave room for border */
    max_h -= 1;
    w_handle = wind_create( 0, pos_x, pos_y, max_w, max_h ); /* use window */
    wind_open( w_handle, pos_x, pos_y, max_w, max_h );
                                            /* WM_REDRAW msg will be waiting */
  do {
        ev_which = evnt_multi(
        1+2+0x10,           /* MU_KEYBD | MU_BUTTON | MU_MESAG */

                /* MU_BUTTON */
        1,              /* number of button clicks to wait for */
        0xF,            /* buttons to wait for -- 0x1 leftmost */
        1,              /* button state to wait for -- 1 pressed */

                /* MU_M1: mouse 1 event */
        0,              /* which event to return on - 0 entr 1 ext */
        0,              /* X pixel position of mouse event rect. */
        0,              /* Y pixel position of mouse event rect. */
        0,              /* width pixels of mouse event rectangle */
        0,              /* height pixels of mouse event rectangle */

                /* MU_M2: mouse 2 event */
        0,              /* */
        0,              /* */
        0,              /* */
        0,              /* */
        0,              /* */

                /* MU_MESAG */
        pevbuff,        /* message buffer -- 16 bytes */

                /* MU_TIMER */
        0,              /* timer value (millisecs): low word */
        0,              /* timer value (millisecs): hi word */

                /* pointers to output vars */
        &junk,          /* X pixel position of mouse at event */
        &junk,          /* Y pixel position of mouse at event */
        &junk,          /* mouse button state: 0x1 leftmost, 0x2...*/
        &junk,          /* kb state: 1-Rshift 2-Lshift 4-Ctrl 8-Aux*/
        &junk,          /* scan code of returned value */
        &junk           /* # times button entered desired state */
  );

  junk = 0;
  if( ev_which & 0x10 ) {           /* Message event? */
     if( evbuff[0] == 20 ) {               /* WM_REDRAW msg? */
        wind_update(BEG_UPDATE);
        graf_mouse( 256,0L );              /* hide mouse */
        pxy[0] = pos_x+1; pxy[1] = pos_y+1;
        pxy[2] = pxy[0] + max_w-2;
        pxy[3] = pxy[1] + max_h-2;
        vsf_style( handle, SOLID );
        vsf_color( handle, WHITE );
        vr_recfl( handle, pxy );       /* set background */
        draw_rc( handle, pos_x+1, pos_y+1, max_w-2, max_h-2 );
        graf_mouse( 257,0L );              /* show mouse */
        wind_update(END_UPDATE);
        junk = 1;                          /* continue looping */
     } else if ( evbuff[0] == 21 ) {       /* WM_TOPPED msg? */
        wind_set( w_handle, WF_TOP, 0,0,0,0 ); /* get control of kb*/
```

```
            } else if ( evbuff[0] == 10 ) {       /* MN_SELECTED msg? */
                meta_out( max_w, max_h );
                rsrc_gaddr( R_TREE, CTRLMENU, &mobj );/* grab menu */
                menu_tnormal( mobj, TITLEOBJ, 1 );
                junk = 1;                         /* continue looping */
            }
        }
    } while( junk );

    wind_close( w_handle );
    wind_delete( w_handle );
    v_clsvwk( handle );                           /* close the workstation */
    appl_exit();                                  /* tell AES we're through */
}
```

Listing 6.1: *CTRLMETA.C: Screen and Metafile Output Routine*

The menu bar allows the user to access the Desk menu, which contains the desk accessory commands, along with the single command for this program. This command causes the drawing on the screen to be output to a metafile. Our "world coordinate system" for the metafile is a simple expansion of the native raster coordinates of the device. The reason we do not use raster coordinates is that raster coordinates generally produce smaller numbers in magnitude than page widths and heights as measured in 1/10 millimeters. This causes the OUTPUT.APP to fail (this is probably a bug). In this example, we multiply the raster coordinates by the constant 10 in order to work around the problem.

DEBUGGING GEM APPLICATIONS

Debugging your GEM application is similar to debugging any application, except that there are a few twists. You can't use print statements (an old trick from BASIC programming days), but if the bug isn't too serious or subtle, you can compile in a call to a routine that you write—called, for example, chkpoint( )—which displays a form alert, along with any information that you want to build into the alert string. Listing 6.2 shows sample chkpoint( ).

```
/* CHKPOINT.C - uses form_alert() to display a debugging message */

#include "portab.h"
#include "machine.h"

#define CHMARK 15
/* CHMARK is first location to replace in following template string */
char *chk_template=
        "[1][Checkpoint|1234567890123456789012345678901234567890][cont]";

VOID chkpoint(msg)      /* for debugging purposes */
char *msg;
{
        char *tt; int cnt;

        for( tt=chk_template+CHMARK, cnt=0; *msg && cnt<40; ++cnt )
                *tt++ = *msg++;
        while( cnt++ < 40 )
                *tt++ = ' ';
        form_alert(1,ADDR(chk_template));
}
```

Listing 6.2: *Sample chkpoint() Routine for Tracing a Program*

You can scatter calls to this routine in your program if you want a more detailed look at how the program is behaving. This technique can be very awkward, however, when your program is very large or very complicated. It also won't help much with subtle bugs that occur as a result of pointer corruption, which is fairly common with C programs.

In order to have the most debugging control, we suggest you use GEMSID on an 8086. This debugger has fairly good symbol handling, and the DRI Developer's Kit contains a program that maps output from other linkers (like DOS LINK) into a symbol format that GEMSID will understand.

A useful set of commands in GEMSID is the graphics extensions to the debugger, all of which start with the letter "Y". The command we use the most is YME (Multi graphics screens Enable), which makes it possible to debug on multiple screens, where the GEMSID debugging output is displayed on a separate screen from your program's graphic output.

Debugging on the Atari is a bit more difficult because the SID68K isn't quite as sophisticated as its 8086 cousin, GEMSID. Both debuggers are documented in their respective versions of the DRI Developer's Kit.

THE GEM BINDINGS

The GEM C binding routines include routines such as **appl_init()** and **v_opnwk()** that you call from your program to invoke GEM functions. We have delayed our discussion of the bindings because most GEM programmers can safely depend on the bindings as a "black box": as long as you use a C compiler recommended in the DRI Developer's Toolkit (Lattice C on the 8086, and DRC on the 68000), you may never need to know any more about bindings. If you are using a different compiler, however, you may need to know more about the bindings in order to use GEM with your compiler. In this section, we will talk about the bindings to give you a better understanding of what they are and how your program uses them.

What Are Bindings?

To understand binding routines, you need to understand how your C program interfaces with the computer. Most C compilers come with a set of library routines, such as open() and printf(), that you call from your program to handle input and output and other well-defined tasks. Many of these routines eventually call the operating system that comes with your computer. Examples of operating systems include PC-DOS, MS-DOS, Concurrent PC-DOS, TOS, and GEMDOS. The operating system controls hardware devices like the screen and the disk drive.

There are two main reasons for using library routines: first, to make it easier for you to write programs, and second, to make it easier to move your program from one computer to another.

GEM should be thought of as an extension to your computer's operating system. GEM provides two kinds of routines: the AES routines correspond roughly to the high-level library routines that come with your compiler, and the VDI routines correspond to the operating system routines. Both of these sets of routines are bound together in one package. The method your program uses to call these routines is very similar to the method your program uses to call the operating system. Your program must fill certain data structures and initialize certain machine registers. It must also invoke an interrupt on the 8086, or, on the 68000, a trap.

In this book, we have almost completely ignored this level of detail because it is hidden in the bindings. The binding routines fill the data structures with the values you pass in the parameters, and then call a small routine written in assembler to set up the machine registers and invoke the interrupt or trap. The DRI Developer's Kit documents the assembler interface in greater detail.

Different Kinds of Bindings

Binding routines come in several flavors. Because each language has its own set of procedure calling conventions, each language has its own set of bindings. Because of different naming conventions between compilers (the most notable differences being between Lattice C and Microsoft C), each different compiler may need its own bindings. What is more, if you program in different memory models on the 8086, you will find that each different memory model has its own binding routines. From each high-level language, the calls to these binding routines looks the same, but the bindings themselves have been constructed differently in order to hide detail that you don't need to know about to write your GEM program.

As of this writing, the DRI Developer's Kit provides bindings only for Lattice C (small model) on the 8086 and for DRC on the 68000. Support for two different 8086 assemblers (DRI's RASM86 and Microsoft's MASM) are included in the Developer's Kit. Binding support for large model Lattice C, Microsoft C, DR Pascal, and Turbo Pascal should be available from the DRI GEM Programmer's Support account (type "GO DRI") on CompuServe.

More Bad News About Bindings

Unfortunately, some of the ordinary library routines provided by your compiler are not compatible with GEM. Specifically, most compiler library routines that print information on the screen assume that the screen is in character mode, whereas GEM uses graphics mode. Another thing to know is that there is a certain amount of overhead associated with the standard library, and the DRI Developer's Kit is oriented for the professional application writer who wants to minimize such overhead. For these reasons, the DRI Developer's Kit provides bindings that do not use any of the compiler's standard library routines.

This incompatibility with your compiler's standard library is possible to overcome, as long as you are willing to avoid using incompatible standard library functions (that is, library functions that write to the console). Another problem you may run into if you try to use your compiler's standard library concerns memory management. Several GEM functions allocate memory from the operating system, most notably **vst_load_fonts( )** and **rsrc_load( )**. Your compiler's standard library initialization routine needs to deallocate memory that is not needed; if it does not, the GEM functions that require memory fail. For an example of how to deallocate memory in the initialization routine, see the startup module examples in the DRI Developer's Kit (PROSTART.A86 or PROSTART.ASM on the 8086, or TCS.S on the Atari).

The fact that DRI's GEM bindings are not easy to use with the standard library for your compiler may be good news to some of you, especially if you are developing a program for sale. Some compilers, most notably Microsoft C, have some peculiar legal requirements for selling programs that have been linked with their standard libraries. Read the license agreement of your compiler very carefully for this kind of pitfall if you plan to sell your program.

A BRIEF SURVEY OF GEM PROGRAMMING TOOLS

This section provides information on where to get the GEM Developer's Kit, as well as a brief survey of some of the compilers that we have used. GEM was designed to work with many different languages and compilers, but as we pointed out in the previous section on bindings, there are no compilers currently available that are well-oriented to GEM programming, especially for less sophisticated programmers. Hopefully, this will be corrected in the near future.

As of this writing, the 8086 version of the GEM Developer's Kit does not include a compiler. It also assumes that you have the DRI Assembler Plus Tools (RASM86 and LINK86) or that you have the Microsoft Assembler (MASM) and LINK. It does contain a complete set of bindings for Lattice C, sample programs, GEMSID, and reference documentation.

To get more information on the GEM Developer's Kit for the IBM PC and 8086 computers, contact

DRI Deutschland
011 49 899 12065

Digital Research, Inc.
60 Garden Ct, Box DRI
Monterey, CA 93942
(408) 649-3896

The Atari GEM Developer's Kit includes DRC-68K, which was an early 68K compiler developed originally by Alcyon, Inc., and originally used on DRI's CP/M-68K. It also contains associated tools (AS68, LO68, and LINK68). Contact Atari for information on the Atari GEM Developer's Kit:

Atari Corp.
1196 Borregas Ave.
Sunnyvale, CA 94088
(408) 745-2000

For the 8086, DRI originally developed most of its GEM applications using Lattice C, Version 2.11, and the bindings in the Developer's Kit can be used directly with this compiler. Lattice C was one of the first C compilers for the 8086 that produced good quality code and that was robust and easy-to-use. For more information on the compiler itself, contact

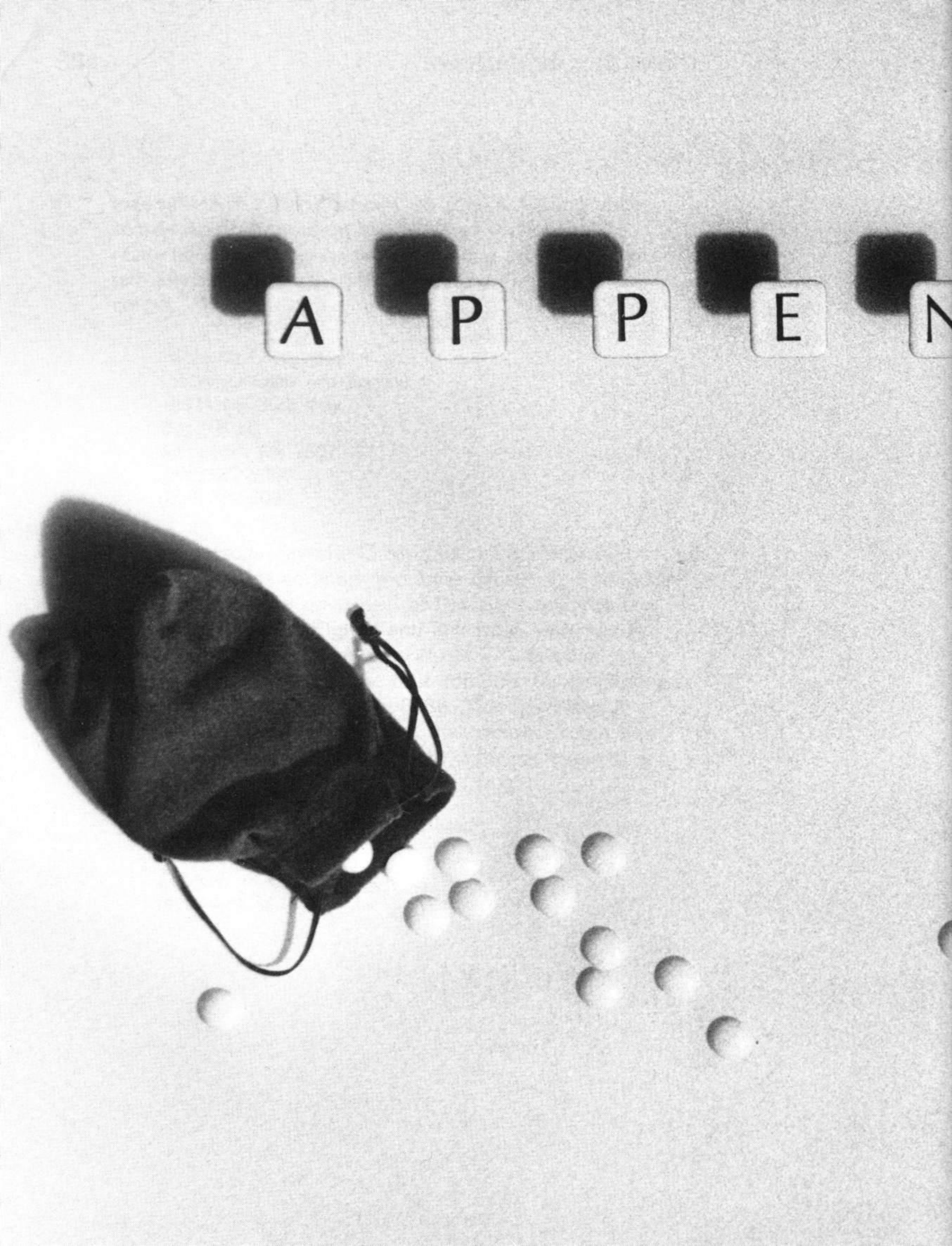
Lattice, Incorporated
P.O. Box 3072
22W600 Butterfield Road
Glen Ellyn, IL 60137
(312) 858-7950 (Sales)
(312) 858-0073 (Support)

We have also used a couple of other compilers. We have found that the Microsoft 3.0 C (MS-C) compiler produces excellent, high-quality code and that the product is fairly easy to use. We needed to modify the bindings, however, because MS-C has a different naming convention for external names than Lattice C (it adds a leading underscore). Another problem we have heard about is the restrictive license agreement for selling programs that contain any code from the MS-C standard library. You can contact Microsoft at

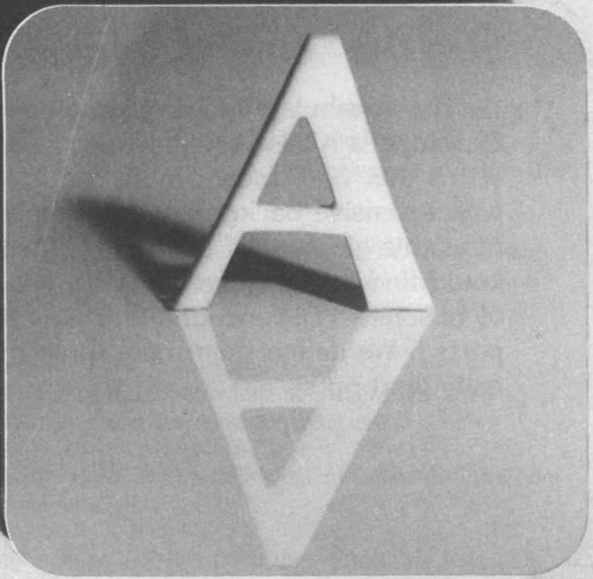
Microsoft Corporation
Customer Sales and Service
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717
(206) 882-8080
(206) 882-8088

Our personal favorite C compiler is the MetaWare High C compiler. This compiler produces excellent, high-quality code. It also has a great deal of flexibility, and you can compile routines in High C and link them with routines compiled by Lattice or Microsoft, as well as other languages like Pascal. It is the best compiler for building professional applications on the 8086. This flexibility, however, is difficult to use, and the compiler requires a hard disk system. We recommend it for professional software developers. High C is available from

MetaWare Inc.
412 Liberty Street
Santa Cruz, CA 95060
(408) 429-META


A P P E N

D I X



GLOSSARY

The vocabulary of GEM programming and the GEM Developer's Kit is very rich with jargon and technical terms that were meant more for systems programmers with extensive backgrounds in graphics programming. We offer this glossary of as many GEM buzzwords as we could find in the Developer's Kit. Some of the terms refer to obsolete VDI functions that may no longer be supported. We define them here mostly to satisfy your curiosity (and ours).

A

Absolute Mode A method of setting text character height in terms of raster or normalized device coordinates using **vst_height( )**; to be compared with *points mode,* where the text height is specified in points (1/72 inch) using **vst_points( )**.

Active Window The window "on top" of any other window on the Desktop; the process owning the active window receives all mouse and keyboard input.

.ACC GEM desk accessory executable file.

Alert A standardized message to the user, consisting (usually) of an icon specifying NOTE, WAIT, or STOP, plus a short message, plus one or more boxes to let the user acknowledge the alert by clicking in one of the boxes.

Aliasing An effect that is the by-product of scaling bit images such as letters. A greater amount of aliasing will cause objects drawn with diagonal lines, such as the letter "N", to display a jagged or staircase effect on the diagonal lines.

.APP GEM application executable file.

Application Environment Services (AES) A part of the resident GEM services containing subroutines that define the user interface. The AES is a set of operations including windows, forms, and menus. See Chapter 2.

Application Id A WORD value returned by the **appl_init( )** call that uniquely identifies each application; can also be used for interprocess communication in sending messages to other GEM processes.

Arrow A form of the cursor, the arrow is displayed on the Desktop and moves in direct relationship with the mouse or pointing device to indicate the focus of the user's interaction with GEM or the GEM application.

Aspect Ratio The ratio of height to width of a rectangle that appears on the graphics device to be a square.

ASSIGN.SYS Data file used by the GEM VDI to determine the configuration of the system.

Attributes Variables set by VDI functions that determine how VDI output functions will display an image. Examples of text attributes that can be set include text color (**vst_color( )**) and size (**vst_height( )**). See also *characteristics.*

B

Bindings The subroutine calls that the programmer uses to invoke GEM or DOS operations. Bindings are the high-level language interface to the functions; they vary from language to language.

BITBLT Bit image block transfer; an action denoting moving part of the bit map to another part of the bit map; see *raster operation.*

Bit Map A bit map is a collection of pixels used to represent an image.

Blocking The act of calling a synchronization routine and not returning from the call until some specified event has occurred; see *polling.*

C

Carp Official mascot of the Digital Research GEM team.

Cell Array A precursor to BITBLT, this was a data structure that was used for producing scaled bit block transfers on raster screens.

The cell array operation in the VDI is not supported on very many common devices, and will probably be removed from the VDI.

Character Baseline Vector An imaginary vector that defines the bottom line for text display. All of the characters in a string of text will be displayed resting on the baseline (descenders, though, will cross the baseline).

Character Snap An RCS term for the optimization of aligning and displaying all text and bit image objects on byte (or character) boundaries. This optimization greatly enhances the time it takes to display the text or bit image.

Characteristics What a device is capable of displaying. Examples of characteristics include the number of addressable units of vertical and horizontal display. See also *attributes.*

Click A mouse technique: the user guides the mouse to a desired location on the screen, presses the button, and (without moving the mouse) releases the button.

Click-drag A mouse technique in which the user positions the mouse, presses the button, and moves the mouse to another location on the screen before releasing the button. See *click.*

Clipping Rectangle A rectangle defining the bounds of the allowable display region. Any graphics output that would occur outside of the clipping rectangle will not be displayed.

Control Array One of the arrays used by the VDI bindings.

Control Point See *window control point.*

Coordinate Scaling The process of transforming coordinates from the current coordinate system (for example, NDC) to the device's coordinate system (for example, RC).

Coordinate Systems A means of specifying the location of graphical objects. Coordinate systems come in several flavors, specifically device-independent (normalized device coordinates or image coordinates) or device-dependent (raster coordinates).

Cursor Point on the screen where the next screen I/O will take place. Usually there is some sort of indicator pointing at this position—for example, an arrow, a flashing block, or an underline.

D

.DEF Resource companion file, produced by Resource Construction Set to provide additional information on resource (.RSC) files; used by Resource Construction Set only.

Default Device Driver First driver named in the ASSIGN.SYS file; must be the largest driver that will be loaded.

Default Directory An operating system term referring to the disk and directory on that disk in which a file is opened or created, unless the disk and/or directory are explicitly specified.

DEFAULT.OPT Data file used by OUTPUT.APP to maintain option settings.

Descenders The lower part of lowercase letters, such as "j" and "q".

Desk Accessory A GEM application that does not take over the entire screen, but uses only a small window. They can be invoked from any running GEM application. At present, the desk accessories are Calculator, Clock, and a Printer Spooler.

Desktop 1) GEM user interface from DRI that uses the office as a metaphor. It provides file/directory information by means of icons. 2) The default display when an application starts up, usually a grey rectangle which fills the screen.

DESKTOP.INF Data file produced by the Desktop which contains the user's Preferences data—for example, which directory is to be displayed when the Desktop starts up, button timing, and so on.

Device Coordinates Coordinates used by a particular device; see *raster coordinates.*

Device Driver Software module that provides the interface between the operating system and the device.

Device Handle See *VDI handle.*

Device Identification Number ID number assigned to a device in the ASSIGN.SYS file. This number is used in the Open Workstation Call to specify which device driver is to be used.

.DFN The replacement extension for .DEF for version 2 of the RCS.

Dialog A special kind of form (built with the RCS and used with the **form_do( )** call) whose purpose is to accept input from the user. A dialog may contain most of the general object types (boxes, text, input fields, bit images, and so on). The RCS will align the display of these objects on character boundaries for performance reasons; see *character snap* and *panel.*

Disabled An object state where the object is drawn faintly, usually to indicate that the object is not available (enabled) for user selection.

Dispatcher Component of GEM that removes the currently running process from the top of the Ready list, places it on the Not Ready list or at the bottom of the Ready list, and moves all the processes on the Ready list up one position. The process on top of the Ready list is currently executing.

DOS The disk operating system; most often equated with PC-DOS (DOS on the IBM PC).

Double-click The physical act of clicking twice on an object or an icon. The Desktop uses this technique as a shortcut for opening an object: that is, if the object is a folder, its contents replace the objects in the window; if it's a data file for an installed application, the application is invoked, and so on.

Drop-down Menu The menus in GEM AES are known as drop-down menus, because as soon as the mouse moves into the menu's screen area, the menu is "dropped" onto the screen. Other similar systems use pull-down menus, where the menus must be click-dragged down.

E

Entry Point In a computer program, an instruction, statement, or procedure that is first to receive control.

Escape A mechanism to indicate special handling; usually used in the context of a special sequence of control characters sent to a device or of a special function call.

Event The occurrence of some well-defined action, such as the user's activation of a window control point or clicking the mouse button inside the application's window.

F

Face Short for "type face"; a particular style of letters used for the display of textual information.

Far Pointer A far pointer consists of a LONG (32-bit) value consisting of two word values, an offset and a segment value, and can address any location in the 8086 physical memory. The GEM AES uses far pointers into resources to maximize flexibility and minimize copying information. See *memory model.*

File Handle The value that is returned from an open or create file DOS function call. The file handle is used by all the other DOS I/O routines to identify a particular file. (You open/create using file names, but you read/write using file handles.)

Fill Pattern A rectangular sequence of bits that is repeated over and over again to fill a specified area. A gray fill pattern, used to fill the Desktop surface, for example, has every other bit set.

form_do( ) An AES function call used to activate forms and dialogs to manage user interaction in a consistent fashion.

Free Strings An RCS term indicating a group of string objects built into a resource file, and thus editable by the RCS. An application that uses free strings can remove any language dependencies in the user interface from the compiled part of the program. This means that an application's resource file can be edited by a nonprogrammer to allow the application's user interface to be tailored for a foreign language.

Full Box A window control point whose activation indicates that the user wants to make the window as large as it can get or that the user wants to return it to its former (nonfull) size.

Function Code A parameter to the VDI or AES that indicates the desired GEM service.

G

.GEM A metafile, containing VDI operations; data file type for GEM DRAW picture files.

GEM Developer's Kit A product from Digital Research that provides you with the necessary tools to build GEM applications, such as the Resource Construction Set, the Icon Editor, and the language bindings for the AES and VDI services.

GEM Programmer Support (GPS) A service offered by Digital Research that gives the GEM programmer a CompuServe account for 24-hour access to GEM programming information, and a hotline number to call for questions still unanswered.

GEMSID A version of Digital Research's SID86 (Symbolic Instruction Debugger) modified to make use of the GEM environment.

GEM Toolkit See *GEM Developer's Kit.*

Generalized Drawing Primitive (GDP) A series of routines in the VDI that displays shapes such as circles, ellipses, and arcs.

Graphic Cursor Usually in the shape of a small diagonal arrow, it is moved about the screen in correspondence with the mouse movements or under keyboard control and is used to indicate a position on the screen. Different shapes for the graphic cursor can be used to indicate program states; for example, an hourglass may indicate that the program is performing a lengthy calculation and will be temporarily unresponsive. The graphic cursor is controlled by the **graf_mouse( )** function.

Graphical Kernel System (GKS) The VDI's predecessor, the GKS was a system of device-independent graphical display functions.

Graphics Command The means of specifying to a graphics device exactly what is to be displayed.

Graphics Device A piece of hardware designed to display pictures or images, or to accept two-dimensional (pointer-oriented) input.

Graphics Device Operating System (GDOS) That portion of the VDI that interprets the requested command and passes it to the appropriate device driver. The GDOS handles translation of normalized device coordinates to raster coordinates, and also handles loading and unloading text fonts.

Graphics Primitives Commands to display fundamental graphics shapes, such as lines, markers, text and rectangles.

GRECT A data structure used to specify the location of a rectangle on the screen.

H

Handle The value returned by the Open Workstation function; used like a file handle, except that the object in question is not a file, but a graphics workstation.

Home Directory See *default directory.*

Hot Spot The location in the mouse form that gets sent to the application when the user presses the button; in the arrow mouse form, the hot spot is in the upper left corner.

I

.ICN GEM Icon Image file; output by Icon Editor, used by Resource Construction Set.

Icon A rectangular sequence of bits that forms a tiny picture.

Icon Editor GEM application that allows a user to create or modify icons.

Image Coordinates Device-independent coordinate system used by an application; other coordinate systems used in GEM programming are normalized device coordinates and raster coordinates. Image coordinates must be transformed to raster coordinates before they are passed into the VDI. Also known as *world coordinates.*

.IMG A bit image file; data file type for GEM Paint image files.

Input Locator A generic term for a graphical input device that returns a two-dimensional quantity, usually a cursor position on the screen.

Input Valuator A generic term for a one-dimensional graphics input device that returns the position of some measuring device on a linear scale. The VDI supported input valuators on some uncommon devices, such as dials on older graphics devices, but you should not count on VDI support for input valuators. The sliders on GEM window control points are examples of AES input valuators.

Invert The process of switching foreground and background colors, usually used to indicate object selection.

J

justification The process of causing a number of objects, usually words, to line up; left justification would cause the leftmost word or object to line up, and right justification does the same for the rightmost word or object.

L

Large Model Large model has large code pointers, large data pointers, and each compiled module can contain up to 64K of static data (as opposed to big model, where all of the compiled modules together must be less than 64K). See *memory model*.

Large Pointer See *far pointer*.

LONG A compiler independent declaration for a 32-bit integer. See *PORTAB.H*.

Long Pointer See *far pointer*.

M

MACHINE.H A set of macros designed to hide certain detail involving processor-dependent operations. For example, MACHINE.H contains macros allowing the programmer to access the high-order byte of a WORD value regardless of the byte ordering of the processor.

.MAP Linker output file listing program modules and where they are located within the executable file.

Medium Model Medium model has large code pointers, thus allowing a program's code to be more than 64K, and small data pointers. See *memory model.*

Memory Model A program organization on the Intel 8086 segmented architecture. The term refers to how the code and data portions of the program are addressed. Each portion of the program can be addressed by short (16-bit) or long (32-bit) pointers. Short pointers are more efficient but can only access up to 64K of code or data, whereas long pointers can access any location in physical memory. Programs using long pointers are, however, less efficient in space and time. See *segmented architecture, big model, compact model, far pointer, large model,* and *small model.*

Menu A command or option selection mechanism provided by GEM; allows the user to choose from a list of text or bit image descriptions of commands or options.

Menu Bar An area of the screen reserved for menu selection. The menu bar is controlled by the process but is independent of any of the process's windows.

Menu Sidebar A portion of a window (as opposed to the screen) set aside by an application for a graphical selection of commands or options.

Message A data structure passed from one process in GEM to another (possibly itself). Many GEM messages contain event information.

Metafile A file containing graphics commands that can be displayed (by OUTPUT.APP) on any of the devices supported by GEM. The GEM Draw application uses metafiles for input and output.

Memory Form Definition Block (MFDB) A VDI data structure whose purpose is to define how a sequence of words in memory gets mapped onto a raster (screen) device.

Mode A program state that governs how the program interprets input from the user.

Mouse Popular pointing device that allows the user to move the cursor around on the screen.

Mouse Button A control button on the mouse device. Once the mouse has been positioned on the screen, one of the mouse buttons can be pressed to select the object pointed at by the mouse. The number of buttons on a mouse may vary. Typically, there are one to three.

Mouse Technique A method of using or operating the mouse; for examples, see *click* and *shift-click*.

N

Normalized Device Coordinates (NDC) A device-independent coordinate system that specifies 32768 horizontal degrees of resolution by 32768 vertical degrees of resolution. Generally, applications will prefer to use their own application-specific image coordinate system, partly because normalized device coordinates don't help much with aspect ratio. Transformation of NDC points to the device adds overhead to the drawing functions.

NDC Space The collection of all possible normalized device coordinates for a given device.

NULLPTR A compiler-independent declaration of a pointer with value 0. See *PORTAB.H*.

O

Object An AES data item used to manage part of the graphics display. Objects are organized recursively into trees, where a display object may contain other objects (or object trees).

Object Tree A recursive, hierarchical organization of objects.

Obspec Short for "object specification"; a LONG (32-bit) field in the object data structure containing additional, object-specific information, such as color, or a pointer to additional information.

Operation Codes (opcodes) The particular bit sequences that specify certain commands, either machine opcodes (processor-specific commands) or graphics opcodes (graphics commands).

OUTPUT.APP A GEM application that transfers the file types .GEM or .IMG to output devices such as a printer, plotter, or film recorder.

P

Panel An RCS term for a collection of objects, similar to a dialog; differs from a dialog because character snap is not enforced, thus allowing fine placement of objects at the cost of increased display time.

Path When dealing with hierarchical file systems (such as DOS), a path or path name is the complete file name that absolutely identifies where in the file tree the file exists. Thus, root/dira/dirb/filea is different from root/dira/dirc/filea, where each directory is specified by the name within slashes (/).

Pel See *pixel.*

Pixel A pixel is a picture element. If the picture is in black-and-white, then a pixel may be just a bit. However, as color is added, the size of a pixel may be measured in several bits, depending on the number of colors.

Point Mode Device-independent method of specifying text character height in 1/72 inch increments known as "points"; see the **vst_points**( ) function.

Polling The action of a program whereby the program asks if some event such as keyboard input has occurred. A more efficient method for GEM programs to get input is called *blocking,* which means calling one of GEM's event handling routines, such as

evnt_multi( ), and letting the system suspend your process until the event has happened.

Polygon A many sided, two-dimensional figure. For instance, the following are polygons: rectangles, squares, pentagons, and duodecahedrons.

PORTAB.H A file containing C language data type declarations. The data type declarations in the file can be modified for different compilers, in order to allow the data types declared in the file to always be the same size. For example, the C data declaration integer may be 16 bits or 32 bits, depending on compiler. The PORTAB.H declaration for WORD can be changed so that WORD always specifies a 16-bit quantity.

Programmer-defined Object A graphical data item (object) that will be drawn by a subroutine supplied by the GEM application programmer. This is essentially an "escape hatch" that allows the programmer to supply custom drawing routines for special objects and still use most of the standard AES routines for standard object display.

Pull-down Menu A type of menu that requires the user to click on the menu area and drag it down. See *drop-down menu.*

R

Radio Button A type of object, considered with a collection of objects, where one and only one of the objects in the collection can be selected at any time.

Raster Area A collection of pixels used to represent a picture or a portion of the display.

Raster Coordinates (RC) Device-specific coordinate system; the finest positions of addressability for the device.

Raster Operations VDI functions that operate on raster areas, usually by copying or transforming them. An example is **vro_cpyfm**( ).

Raster Functions See *raster operations.*

RC Space The collection of all raster coordinates of a device.

RCS See *Resource Construction Set.*

Rectangle 1) A portion of the screen, specified either by diagonally opposite corners of a rectangular region or by the upper left corner plus a width and height. 2) A data structure containing the information necessary to specify a portion of the screen.

Rectangle List The series of rectangles, returned by succesive calls to **wind_get( )**, that define the portion of the screen that needs to be redrawn.

Resource A collection of data objects used by the AES functions. Some examples of resources include forms, dialogs, and alerts. A resource allows the data objects to be modularized (relative to the programmer's code) and manipulated separately by the Resource Construction Set. This allows the data displays to be designed separately from the program. It also allows the data displays to be changed independently of the program (thus allowing a nonprogrammer to translate the program's displays into a foreign language, for example).

Resource File A collection of resources, grouped into a single file, usually having the file type .RSC.

Resource Construction Set A program in the GEM Developer's Kit that allows the programmer to graphically construct the components in the resource file.

Root The first object in an object tree. Also known as a "root node." See *tree.*

.RSC Resource file, output of Resource Construction Set, containing GEM resource and image information to be loaded by an application.

Rubber Box A graphical display provided by the AES to indicate a changeable area of the screen. The rubber box is used by the Window Manager, for example, to give the user the currently selected screen size when the user is resizing the window. A rubber box is also used by the Desktop to allow the user to select a number of objects in the window. The rubber box is implemented with the **graf_rubberbox( )** AES call.

Run-time Library (RTL) General-purpose subroutines. Usually a programming language like C provides an extensive number of run-time library routines that the programmer calls explicitly to perform tasks such as disk I/O. The run-time library also contains subroutines that the compiler calls implicitly, rather than generating in-line code, to handle more complicated tasks such as multiplication and division. Since a run-time library is usually general-purpose, its routines can contain code that would never be used by the application. (For example, the **printf( )** function in C must be able to print floating point numbers, which may never be used by your program but which would require large amounts of code.) The GEM Developer's Kit provides special purpose run-time libraries in place of the more general-purpose language run-time libraries.

S

Scrap Information passed between applications that is stored on disk and manipulated by GEM programs through the Scrap Library Manager.

Screen Manager The part of the AES that handles the GEM windows; provides functions for the application to create, open, move, and size windows as well as a standardized mechanism whereby the user can specify window operations to the application.

Scroll Bar A window control point (vertical or horizontal) contained in the window border (rightmost for vertical, bottom for horizontal). The scroll bar, along with its slider, gives the user an idea of how much the window displays of the total possible display, and approximately where the window displays into the total possible display.

Segmented Architecture An addressing organization of a computer. A segmented architecture divides up the computer's main memory into pieces (segments) that can be used independently of other pieces. For example, the computer may have certain parts of memory dedicated for program code and other parts for program data. In a segmented architecture, these pieces are strictly limited in size (64K on the Intel 8086 architecture). A different addressing method is called a "flat addressing space," whereby all code and data is referenced uniformly (as in the Motorola 68000 architecture).

Select A user action whereby the user chooses which object (for example, an icon) is to be manipulated, or which action (for example, a menu item) is to be performed. When the user has clicked on the object, the application can choose to change its state to *selected* to indicate the selection.

Selected An object state in which the object is highlighted by being drawn with its foreground and background colors reversed.

Shell The part of the operating system that interprets user requests; usually the shell accepts typed commands from the user and executes a program. In GEM, the shell can be used to run a non-GEM program (with the **shel_write**( ) function).

Shift-drag A mouse technique whereby the user positions the mouse in a desired area (object), presses the mouse button, moves the pointer to some new location, and releases the mouse button. The user uses this technique, for example, to drag a file icon to the trash can to indicate file deletion.

Sidebar See *menu sidebar.*

Size Box A window control point allowing the user to request that the size of the display window be changed.

Slider A rectangular-shaped window control point contained in the vertical or horizontal scroll bar. Under programmer control, the slider indicates the relative amount of total possible display area currently on the screen (this is indicated by the size of the slider, relative to the scroll bar), as well the location in the total display area of the screen display (this is indicated by the location of the slider relative to the scroll bar). The slider can be manipulated by the user to indicate a request for moving the screen display to another portion of the total display area.

Small Model Small code and data pointers, where code and data are limited to 64K each; the most efficient programming model for the Intel 8086 architecture. See *memory model.*

Software Performance Report (SPR) A form to be submitted to Digital Research describing incorrect or anomalous system program behavior; a program bug report.

Stack A LIFO (last in, first out) data structure manipulated implicitly from the C language and containing the return addresses for functions and subroutines, as well as any local storage allocated within a function. A stack is often compared to the push-down, pop-off dish storage device commonly used in cafeterias.

Start-up Module The routine that first receives control on program initialization (start up). Usually written in assembler code, the start-up module sets up the stack and heap, and calls the main user entry point.

.SYM Linker output file listing symbols and their locations within the executable file.

T

Timer An AES mechanism allowing the programmer to schedule certain events. The timer is usually used to generate some well-defined response if no other user-generated event occurs. The timer is also used to allow the program to sample the values of the mouse or the keyboard at a certain well-defined rate in order to perform graphics input.

Title Bar A window control point, found in the top border of the window, which contains a string value supplied by the program. The title bar is also used as the place from which the user click-drags in order to move the window.

Transformation Mode The method specified for translating the current coordinate system (NDC or RC) to the device's coordinate system. In raster coordinates (RC), the transformation mode is 1 to 1—that is, there is no transformation. In normalized device coordinates (NDC), the coordinates must be mapped (transformed) into the device's coordinate system.

Trash Can A graphical icon denoting a deletion operation. The GEM Desktop contains a trash can to which the user can drag objects (other icons) on the screen to indicate that the objects represented by the icons are to be deleted.

Tree A linked data structure that defines a hierarchical organization of the data. In GEM, the tree structure is used to display groups of related graphical objects, where each object in the hierarchy is assumed to completely contain (graphically enclose) all of the objects that are below it in the hierarchy. This relation is called a "visual hierarchy" in the Resource Construction Set. Another fact about trees is that all objects below the root node of the tree are located relative to the root node; this makes it very simple to relocate the tree simply by changing the position of the root node.

U

Update Region The part of a program that updates the graphical images on the screen. This part of the program needs to have some method of reserving the screen for itself in order to keep from conflicting with other sequences of VDI commands. The update region is delimited by calls to the **wind_update( )** function.

User-defined Object See *programmer-defined object.*

User Interface The presentation part of a computer and its application that the user sees and interacts with, including the screen display, keyboard, mouse, and any other physical device that the user either receives information from or uses to input information to the computer. The user interface also includes the software that controls these interactions.

V

Valuator See *input valuator.*

Validation The process of checking to see whether user input follows certain criterion, such as being a well-formed number. Validation is provided by the Form library function **form_do( )**.

VDI Handle The value returned by the VDI function **v_opnvwk( )**, which uniquely identifies a workstation.

Virtual Device Interface (VDI) A set of routines that allow display commands to be sent to graphic devices in a device-independent manner, thereby enabling the programmer, for example, to write a single routine that would draw a figure on a screen or on a graphics printer.

Virtual Workstation A graphics device and its attributes that can be modified independently of other virtual workstations open on a physical workstation. Virtual workstations constitute a method of sharing a physical device between several independent programs or subprograms. See *workstation.*

Visual Hierarchy The relationship of visual containment for graphics objects; see *tree.*

W

Window A region of the screen managed in a standard way. Windows facilitate management of the display, giving the user the tools to customize his display in a manner that is consistent across applications.

Window Control Point One of several components of a window, a window control point is displayed in the border of the window and provide a standard graphical way of specifying window operations to the application. Some common window control points include CLOSER and MOVER.

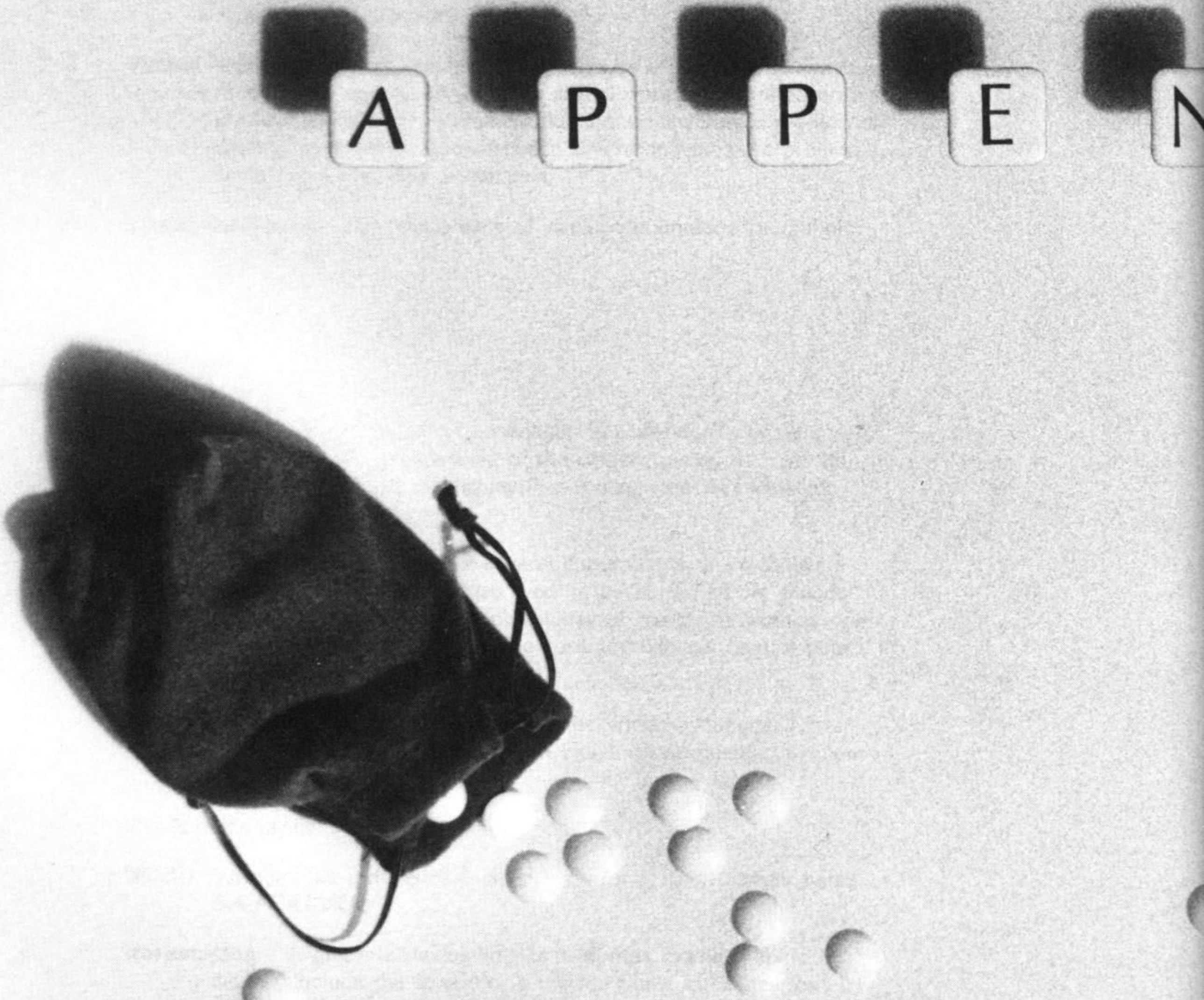
Window Handle The value returned by the Window Manager function **wind_create**( ) to identify a window for succeeding window operations.

Window Manager See *Screen Manager.*

WORD A compiler independent declaration for a 16-bit integer value. See *PORTAB.H.*

Workstation A graphics device and its attributes. Some graphic devices include the screen or a printer. Some attributes for workstations include line color, background color, and type size. A workstation is to graphics as a file is to disk I/O. See *virtual workstation.*

World Coordinates See *image coordinates.*


APPEN

D I X B

AES AND VDI QUICK REFERENCE GUIDE

To give you a quick way of looking up a function's parameters, this appendix contains an alphabetical list of all the functions of the GEM bindings covered in this book, along with parameter declarations.

```
WORD appl_exit( )

WORD appl_find( pname )
    LONG pname;

WORD appl_init( )

WORD appl_read( rwid, length, pbuff )
    WORD rwid, length;
    LONG pbuff;

WORD appl_tplay( tbuffer, tlength, tscale )
    LONG tbuffer;
    WORD tlength, tscale;

WORD appl_trecord( tbuffer, tlength )
    LONG tbuffer;
    WORD tlength;

WORD appl_write( rwid, length, pbuff )
    WORD rwid, length;
    LONG pbuff;

WORD evnt_button( clicks, mask, state, pmx, pmy, pmb, pks)
    WORD clicks, *pmx, *pmy, *pmb, *pks;
    UWORD mask, state;

WORD evnt_dclick( rate, setit )
    WORD rate, setit;

UWORD evnt_keybd( )

WORD evnt_mesag( pbuff )
    LONG pbuff;
```

```
WORD evnt_mouse( flags, x, y, width, height, pmx, pmy, pmb,
                 pks )
    WORD flags, x, y, width, height, *pmx, *pmy, *pmb, *pks;

WORD evnt_multi(flags, bclk, bmsk, bst, m1flags, m1x, m1y, m1w,
                m1h, m2flags, m2x, m2y, m2w, m2h, mepbuff,
                tlc, thc, pmx, pmy, pmb, pks, pkr, pbr )
    UWORD flags, bclk, bmsk, bst, m1flags, m1x, m1y, m1w, m1h;
    UWORD m2flags, m2x, m2y, m2w, m2h, tlc, thc,
    UWORD *pmx, *pmy, *pmb, *pks, *pkr, *pbr;
    LONG mepbuff;

WORD evnt_timer( locnt, hicnt )
    UWORD locnt, hicnt;

WORD form_alert( defbut, astring )
    WORD defbut;
    LONG astring;

WORD form_center( tree, pcx, pcy, pcw, pch )
    LONG tree;
    WORD *pcx, *pcy, *pcw, *pch;

WORD form_dial( dtype, ix, iy, iw, ih, x, y, w, h )
    WORD dtype, ix, iy, iw, ih, x, y, w, h;

WORD form_do( form, start )
    LONG form;
    WORD start;

WORD form_error( errnum )
    WORD errnum;

WORD fsel_input( pipath, pisel, pbutton )
    LONG pipath, pisel;
    WORD *pbutton;

VOID graf_dragbox( w, h, sx, sy, xc, yc, wc, hc, pdx, pdy )
    WORD w, h, sx, sy, xc, yc, wc, hc, *pdx, *pdy;

VOID graf_growbox( orgx, orgy, orgw, orgh, x, y, w, h )
    WORD orgx, orgy, orgw, orgh, x, y, w, h;
WORD graf_handle( pwchar, phchar, pwbox, phbox )
     WORD *pwchar, *phchar, *pwbox, *phbox;

VOID graf_mbox( w, h, srcx, srcy, dstx, dsty )
     WORD w, h, srcx, srcy, dstx, dsty;

WORD graf_mkstate( pmx, pmy, pmstate, pkstate )
     WORD *pmx, *pmy, *pmstate, *pkstate;

WORD graf_mouse( m_number, m_addr )
     WORD m_number;
     LONG m_addr;

VOID graf_rubbox( xorigin, yorigin, wmin, hmin, pwend, phend )
     WORD xorigin, yorigin, wmin, hmin, *pwend, *phend;

VOID graf_shrinkbox( orgx, orgy, orgw, orgh, x, y, w, h )
     WORD orgx, orgy, orgw, orgh, x, y, w, h;

VOID graf_slidebox( tree, parent, obj, isvert )
     LONG tree;
     WORD parent, obj, isvert;

VOID graf_watchbox( tree, obj, instate, outstate )
     LONG tree;
     WORD obj;
     UWORD instate, outstate;

WORD menu_bar( tree, showit )
     LONG tree;
     WORD showit;

WORD menu_icheck( tree, itemnum, checkit )
     LONG tree;
     WORD itemnum, checkit;

WORD menu_ienable( tree, itemnum, enableit )
     LONG tree;
     WORD itemnum, enableit;
WORD menu_register( pid, pstr )
    WORD pid;
    LONG pstr;

WORD menu_text( tree, inum, ptext )
    LONG tree, ptext;
    WORD inum;

WORD menu_tnormal( tree, titlenum, normalit )
    LONG tree;
    WORD titlenum, normalit;

WORD objc_add( tree, parent, child )
    LONG tree;
    WORD parent, child;

WORD objc_change( tree, drwob, dpth, xc, yc, wc, hc, newstate,
                  redraw )
    LONG tree;
    WORD drwob, dpth, xc, yc, wc, hc, newstate, redraw;

WORD objc_delete( tree, delob )
    LONG tree;
    WORD delob;

WORD objc_draw( tree, drawob, depth, xc, yc, wc, hc )
    LONG tree;
    WORD drawob, depth, xc, yc, wc, hc;

WORD objc_edit( tree, obj, inchar, idx, kind )
    LONG tree;
    WORD obj, inchar, *idx, kind;

WORD objc_find( tree, startob, depth, mx, my )
    LONG tree;
    WORD startob, depth, mx, my;

WORD objc_offset( tree, obj, poffx, poffy )
    LONG tree;
    WORD obj, *poffx, *poffy;
WORD objc_order( tree, mov_obj, newpos )
     LONG tree;
     WORD mov_obj, newpos;

WORD rsrc_free( )

WORD rsrc_gaddr( rstype, rsid, paddr )
     WORD rstype, rsid;
     LONG *paddr;

WORD rsrc_load( rsname )
     LONG rsname;

WORD rsrc_obfix( tree, obj )
     LONG tree;
     WORD obj;

WORD rsrc_saddr( rstype, rsid, lngval )
     WORD rstype, rsid;
     LONG lngval;

WORD scrp_read( pscrap )
     LONG pscrap;

WORD scrp_write( pscrap )
     LONG pscrap;

WORD shel_envrn( ppath, psrch )
     LONG ppath, psrch;

WORD shel_find( ppath )
     LONG ppath;

WORD shel_get( pbuffer, len )
     LONG pbuffer;
     WORD len;

WORD shel_put( pdata, len )
     LONG pdata;
     WORD len;
WORD shel_read( pcmd, ptail )
    LONG pcmd, ptail;

WORD shel_write( doex, isgr, iscr, pcmd, ptail )
    WORD doex, isgr, iscr;
    LONG pcmd, ptail;

VOID v_arc( handle, xc, yc, rad, sang, eang )
    WORD handle, xc, yc, rad, sang, eang;
    Available for metafiles.

VOID v_bar( handle, xy )
    WORD handle, xy[ ];
    Available for metafiles.

VOID v_circle( handle, xc, yc, rad )
    WORD handle, xc, yc, rad;
    Available for metafiles.

VOID v_clrwk( handle )
    WORD handle;
    Available for metafiles.

VOID v_clsvwk( handle )
    WORD handle;

VOID v_clswk( handle )
    WORD handle;
    Available for metafiles.

VOID v_ellarc( handle, xc, yc, xrad, yrad, sang, eang )
    WORD handle, xc, yc, xrad, yrad, sang, eang;
    Available for metafiles.

VOID v_ellipse( handle, xc, yc, xrad, yrad )
    WORD handle, xc, yc, xrad, yrad;
    Available for metafiles.

VOID v_ellpie( handle, xc, yc, xrad, yrad, sang, eang)
    WORD handle, xc, yc, xrad, yrad, sang, eang;
    Available for metafiles.
```

VOID **v_fillarea**(handle, count, xy)
    WORD handle, count, xy[ ];
    Available for metafiles.

VOID **v_gtext**(handle, x, y, string)
    WORD handle, x, y;
    BYTE *string;
    Available for metafiles.

VOID **v_justified**(handle, x, y, string, length, word_space,
            char_space)
    WORD handle, x, y, length, word_space, char_space;
    BYTE string[ ];
    Available for metafiles.

VOID **v_opnvwk**(work_in, handle, work_out)
    WORD work_in[ ], *handle, work_out[ ];

VOID **v_opnwk**(work_in, handle, work_out)
    WORD work_in[ ], *handle, work_out[ ];
    Available for metafiles.

VOID **v_pieslice**(handle, xc, yc, rad, sang, eang)
    WORD handle, xc, yc, rad, sang, eang;
    Available for metafiles.

VOID **v_pline**(handle, count, xy)
    WORD handle, count, xy[ ];
    Available for metafiles.

VOID **v_pmarker**(handle, count, xy)
    WORD handle, count, xy[ ];
    Available for metafiles.

VOID **v_rbox**(handle, xy)
    WORD handle, xy[ ];
    Available for metafiles.

VOID **v_rfbox**(handle, xy)
    WORD handle, xy[ ];
    Available for metafiles.

VOID **vm_filename**(handle, filename)
WORD handle;
BYTE *filename;
Available for metafiles.

VOID **vq_extnd**(handle, owflag, work_out)
WORD handle, owflag, work_out[ ];

VOID **vqt_extent**(handle, string, extent)
WORD handle, extent[ ];
BYTE string[ ];

VOID **vqt_font_info**(handle, minADE, maxADE, dists, maxwidth, effects)
WORD handle, *minADE, *maxADE, dists[ ], *maxwidth, effects[ ];

WORD **vqt_width**(handle, character, cell_width, left_delta, right_delta)
WORD handle, *cell_width, *left_delta, *right_delta;
BYTE character;

VOID **vr_recfl**(handle, xy)
WORD handle, *xy;
Available for metafiles.

VOID **vr_trnfm**(handle, srcMFDB, desMFDB)
WORD handle;
MFDB *srcMFDB, *desMFDB;

VOID **vro_cpyfm**(handle, wr_mode, xy, srcMFDB, desMFDB)
WORD handle, wr_mode, xy[ ];
MFDB *srcMFDB, *desMFDB;

VOID **vrt_cpyfm**(handle, wr_mode, xy, srcMFDB, desMFDB, index)
WORD handle, wr_mode, xy[ ], *index;
MFDB *srcMFDB, *desMFDB;

VOID **vs_clip**(handle, clip_flag, xy)
WORD handle, clip_flag, xy[ ];
Available for metafiles.

WORD **vsf_color**(handle, index)
    WORD handle, index;
    Available for metafiles.

WORD **vsf_interior**(handle, style)
    WORD handle, style;
    Available for metafiles.

WORD **vsf_perimeter**(handle, per_vis)
    WORD handle, per_vis;
    Available for metafiles.

WORD **vsf_style**(handle, index)
    WORD handle, index;
    Available for metafiles.

WORD **vsl_color**(handle, index)
    WORD handle, index;
    Available for metafiles.

VOID **vsl_ends**(handle, beg_style, end_style)
    WORD handle, beg_style, end_style;
    Available for metafiles.

WORD **vsl_type**(handle, style)
    WORD handle, style;
    Available for metafiles.

WORD **vsl_width**(handle, width)
    WORD handle, width;
    Available for metafiles.

WORD **vsm_color**(handle, index)
    WORD handle, index;
    Available for metafiles.

WORD **vsm_height**(handle, height)
    WORD handle, height;
    Available for metafiles.

WORD **vsm_type**(handle, symbol)

```
    WORD handle, symbol;
    Available for metafiles.

VOID vst_alignment( handle, hor_in, vert_in, hor_out, vert_out )
    WORD handle, hor_in, vert_in, *hor_out, *vert_out;
    Available for metafiles.

WORD vst_color( handle, index )
    WORD handle, index;
    Available for metafiles.

WORD vst_effects( handle, effect )
    WORD handle, effect;
    Available for metafiles.

WORD vst_font( handle, font )
    WORD handle, font;
```

The following functions are available for metafiles.

```
VOID vst_height( handle, hght, ch_wdth, ch_hght, cell_wdth,
                 cell_hght )
    WORD handle, hght, *ch_wdth, *ch_hght, *cell_wdth,
        *cell_hght;
    Available for metafiles.

WORD vst_load_fonts( handle, select )
    WORD handle, select;
    Available for metafiles.

WORD vst_point( handle, point, ch_wdth, ch_hght, cell_wdth,
                cell_hght )
    WORD handle, point, *ch_wdth, *ch_hght, *cell_wdth,
        *cell_hght;
    Available for metafiles.

VOID vst_unload_fonts( handle, select )
    WORD handle, select;
    Available for metafiles.
```

WORD **vswr_mode**(handle, mode)
    WORD handle, mode;
    Available for metafiles.

WORD **wind_calc**(wctype, kind, x, y, w, h, px, py, pw, ph)
    WORD wctype, x, y, w, h, *px, *py, *pw, *ph;
    UWORD kind;

WORD **wind_close**(w_handle)
    WORD w_handle;

WORD **wind_create**(kind, wx, wy, ww, wh)
    UWORD kind;
    WORD wx, wy, ww, wh;

WORD **wind_delete**(w_handle)
    WORD w_handle;

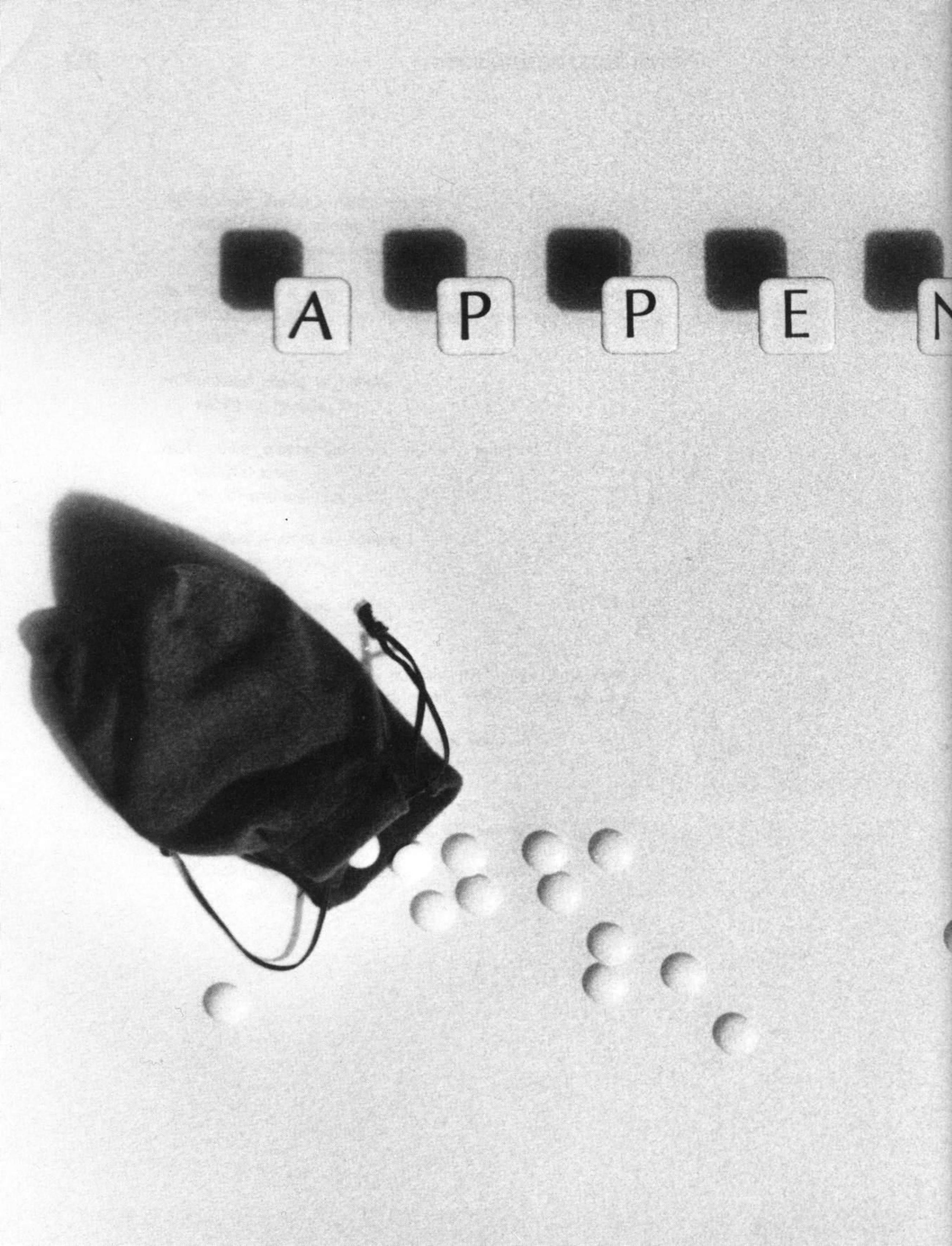
WORD **wind_find**(mx, my)
    WORD mx, my;

WORD **wind_get**(w_handle, w_field, pw1, pw2, pw3, pw4)
    WORD w_handle, w_field, *pw1, *pw2, *pw3, *pw4;

WORD **wind_open**(w_handle, wx, wy, ww, wh)
    WORD w_handle, wx, wy, ww, wh;

WORD **wind_set**(w_handle, w_field, w2, w3, w4, w5)
    WORD w_handle, w_field, w2, w3, w4, w5;

WORD **wind_update**(typ_update)
    WORD typ_update;


APPEN

D I X C

RESOURCE CONSTRUCTION SET TUTORIAL

In this appendix we are going to build the DEMOMENU object tree in a step-by-step fashion. With the information you get from doing this, you should be able to build the other object trees. We assume that you have some knowledge of how to use the Resource Construction Set (RCS). This tutorial does not presume to be the final lesson on using the RCS.

In order to duplicate this tutorial, you must be running the GEM Desktop and be in a directory that has the RCS. We used the second version of the RCS called RCS2. Since the RCS changed a great deal from version 1 to version 2 of GEM, this tutorial will be of minimal use if you have the older version.

1 Double-click on the RCS2 icon, which loads the RCS.

2 Move the mouse cursor to the bottom part of the screen to the Parts Box. To see what the Parts Box looks like, see Figure 6.5.

3 Drag the MENU part to the work area, and release the button. A dialog requesting that you enter a name for the new tree should appear.

4 Backspace over the default name ("TREE1"), and enter "DEMOMENU". Either click on the OK button or press the Enter key.

5 Double-click on the MENU part in the work area.

6 The work area will change and show the menu bar with just the Desk and File menus on top and a darker window area underneath them.

7 Double-click on the the Desk menu name.

8 An "Edit Unformatted String Object" dialog (EUSO) should appear. Move the text cursor (the vertical bar, ¦) to the optional object name field by either pressing the down cursor key or by moving the mouse to the position just after the "Object Name:" field and by then clicking the mouse button. Enter "DEMODESK" and press Return. The EUSO will disappear.

9 Double-click on the Desk menu item. The menu will drop showing "Your message here" and six empty slots for the desk accessory's names.

10 Double-click on the "Your message here" item.

11 A EUSO appears asking for a Text and a Name. Backspace over "Your message here" and enter "About GEM Demo . . .". Then move the cursor down and enter "DEMOINFO" for the optional name.

12 Double-click on the File title (not the RCS File menu, but the one in the work area of the RCS).

13 An EUSO appears. Enter the optional name of "DEMOFILE" and press Return.

14 Click on the Quit item in File menu. Drag on the small dark rectangle in the bottom right corner to the left, so that the you expose a corner of the underlying box. Release. Drag the exposed corner down and to the right, until you have resized the File menu box. The largest string that we are going to add in this box is "Save As . . .". If you don't make a box large enough to hold the largest string, the RCS will display an alert complaining about the autosizing violating the visual hierarchy. Click on the Go Ahead button, and resize the large box later.

15 Drag the Quit item to the bottom of the large box. Click in center of Quit box. An open hand will appear during the drag operation.

16 Drag an ENTRY Part to the top of the large box. Release.

17 Shift-click on the new ENTRY in the large box (hold the Shift key down *before* clicking). Drag the cursor down below the first ENTRY. Release the button. A second ENTRY should appear. Repeat until there are four ENTRYs in the File box. You could also simply repeat step 16, four times, instead of shift-clicking.

18 Double-click on the first ENTRY. An EUSO should appear. Backspace over "ENTRY", and enter "Load". Type in "DEMOLOAD" for the object name, and return.

19 Repeat the last step for each ENTRY, entering

Text: "Save", Optional Name: "DEMOSAVE"
Text: "Save As . . ." Optional Name: "DEMOSVAS"
Text: "Abandon" Optional Name: "DEMOABAN"

20 Click on the Quit item and name it "DEMOQUIT".
We have entered all the File menu items.

21 Click on the work area to close the File menu.

22 Drag a TITLE part next to the File menu. Release.

23 Double click on the TITLE part. An EUSO appears. Backspace over "TITLE" and enter "Options" for the text and "DEMOOPTS" for a name.

24 Click on Options menu. Click on the empty box underneath Options name.

25 Size this box so that we can fit in the largest string—"Pencil/Eraser Selection".

26 Drag an ENTRY part to the Options menu box. Repeat until three ENTRYs are in the Options menu.

27 Double-click on the first ENTRY, and enter the text of "Pencil/ Eraser Selection" and the name of "DEMOPENS".
Now change the string in the second ENTRY to "———————————", and no name.
Finish with the last ENTRY and change the text string to "Erase Picture" with a name of "DEMOERAP".

28 Now that we have entered all the data, we must align the entries. Select the three entries, move the cursor to the far left where the Tools are displayed. Click on the icon showing all the arrows (column 1, row 3). A menu of different alignments should appear. Select the "Align Left".
Now we must set the state and flags field for these objects.

29 Click on the work area to clear the current selection. Click on the first item ("Pencil/Eraser. . .") and then move to the far left again. Select the icon with the "A", (column 2, row 3). A large menu should appear with two halves. The topmost half sets the object flag and the bottom half sets the object state. For the DEMOPENS object, click on the "Selectable" item. The state is left alone.

30 Click on the item consisting of the row of dashes, and set its state to "Disabled".

31 Click on DEMOERAP item, and make its flag "Selectable".
We are finished with the Options menu. Now let's go set the flags and states of the File menu.

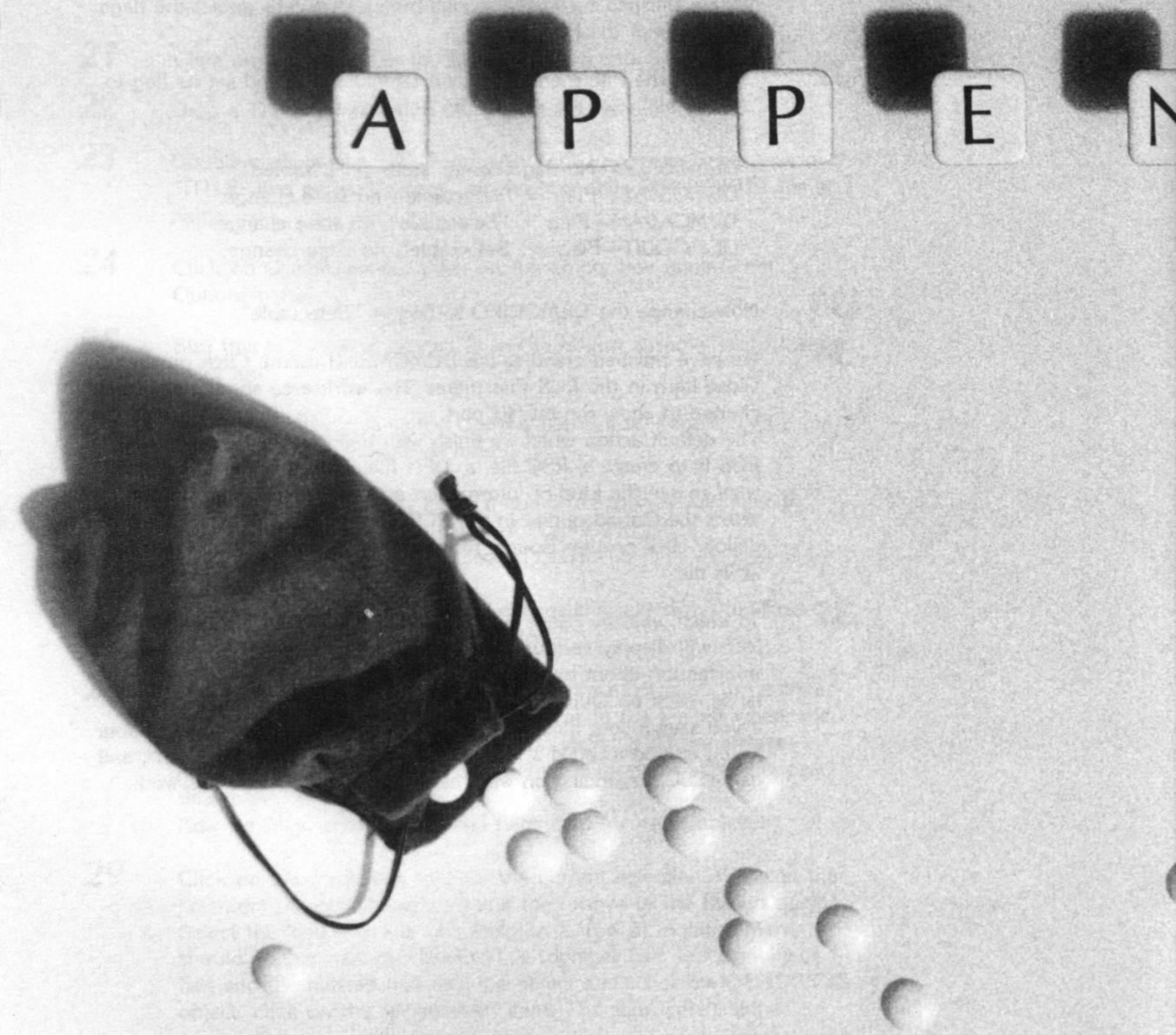
32 Click on the File menu. Click on DEMOLOAD, and set its flag to "Selectable". Set the rest of the items as follows:

DEMOSAVE—No flag change, state = "Disabled".
DEMOSVAS—Flag = "Selectable", no state change.
DEMOABAN—Flag = "Selectable", no state change.
DEMOQUIT—Flag = "Selectable", no state change.

33 Now change the DEMOINFO to flag = "Selectable".

34 We have finished creating the DEMOMENU menu. Click on the Close item in the RCS File menu. The work area should clear and change to show the MENU part.
The default action when we finish with building something in the RCS is to create a .RSC file, a .DFN file, and a .H file. If you also want to see the kind of information as in Appendix G, you must select the Output option in the RCS Global menu. In the displayed dialog, click on the "Source file for resource" button to create .RSH file.

35 To finish, use the "Save As . . ." item in the RCS File menu. The RCS will display various messages telling you that it is writing out information about objects, TEDINFOs, and so on. After it is finished, click on Quit in the File menu to leave the RCS.
If you search your directory, you will find at least three new files (four if you elected to build a .RSH file). They are *name*.H, *name*.DFN, and *name*.RSC, depending on what name you chose to save this work.


A P P E N

APPENDIX B

LISTING OF DEMO

```
#define mc68k 0
/****************************************************************************
*                               PORTAB.H
*
*         C P / M   C   R U N   T I M E   L I B   H E A D E R   F I L E
*
*         -------------------------------------------------------------
*       Copyright 1982 by Digital Research Inc.  All rights reserved.
*       This is an include file for assisting the user to write portable
*       programs for C.
****************************************************************************/
#define UCHARA 1                                /* if char is unsigned      */
/*      Standard type definitions                                           */
#define BYTE    char                            /* Signed byte              */
#define BOOLEAN int                             /* 2 valued (true/false)    */
#define WORD    int                             /* Signed word (16 bits)    */
#define UWORD   unsigned int                    /* unsigned word            */

#define LONG    long                            /* signed long (32 bits)    */
#define ULONG   long                            /* Unsigned long            */

#define REG     register                        /* register variable        */
#define LOCAL   auto                            /* Local var on 68000       */
#define EXTERN  extern                          /* External variable        */
#define MLOCAL  static                          /* Local to module          */
#define GLOBAL  /**/                            /* Global variable          */
#define VOID    /**/                            /* Void function return     */
#define DEFAULT int                             /* Default size             */

#ifdef UCHARA
#define UBYTE   char                            /* Unsigned byte            */
#else
#define UBYTE   unsigned char                   /* Unsigned byte            */
#endif
/****************************************************************************/
/*      Miscellaneous Definitions:                                          */
/****************************************************************************/
#define FAILURE (-1)                    /*      Function failure return val */
#define SUCCESS (0)                     /*      Function success return val */
#define YES     1                       /*      "TRUE"                      */
#define NO      0                       /*      "FALSE"                     */
#define FOREVER for(;;)                 /*      Infinite loop declaration   */
#define NULL    0                       /*      Null pointer value          */
#define NULLPTR (char *) 0              /*                                  */
#define EOF     (-1)                    /*      EOF Value                   */
#define TRUE    (1)                     /*      Function TRUE  value        */
#define FALSE   (0)                     /*      Function FALSE value        */


/*      MACHINE.H               09/29/84-02/08/85       Lee Lorenzen    */


#define PCDOS   1       /* IBM PC DOS */
#define CPM     0       /* CP/M version 2.2 */

#define HILO 0          /* how bytes are stored */

#ifndef I8086
#define I8086   1       /* Intel 8086/8088 */
#endif
```

```
#define MC68K   0          /* Motorola 68000 */

#define ALCYON  0          /* Alcyon C Compiler */

#define ALPHA   1          /* if character screen  */

#define LINKED  0          /* if desktop linked with GEM    */

#define UNLINKED 1


                                                        /* in OPTIMIZE.C          */
EXTERN BYTE       *strcpy();
EXTERN BYTE       *strcat();
EXTERN BYTE       *strscn();
                                                        /* in LARGE.A86           */
                                                        /* return length of       */
                                                        /*   string pointed at    */
                                                        /*   by long pointer      */
EXTERN WORD       LSTRLEN();
                                                        /* copy n words from      */
                                                        /*   src long ptr to      */
                                                        /*   dst long ptr i.e.,   */
                                                        /*   LWCOPY(dlp, slp, n)*/
EXTERN WORD       LWCOPY();
                                                        /* copy n words from      */
                                                        /*   src long ptr to      */
                                                        /*   dst long ptr i.e.,   */
                                                        /*   LBCOPY(dlp, slp, n)*/
EXTERN BYTE       LBCOPY();
                                                        /* move bytes into wds*/
                                                        /*   from src long ptr to*/
                                                        /*   dst long ptr i.e.,   */
                                                        /*   until a null is      */
                                                        /*   encountered, then    */
                                                        /*   num moved is returned*/
                                                        /*   LBWMOV(dwlp, sblp)*/
EXTERN WORD       LBWMOV();

EXTERN WORD       LSTCPY();
                                                        /* coerce short ptr to    */
                                                        /*   low word  of long    */
#define LW(x) ( (LONG)((UWORD)(x)) )
                                                        /* coerce short ptr to    */
                                                        /*   high word  of long   */
#define HW(x) ((LONG)((UWORD)(x)) << 16)
                                                        /* return low word of     */
                                                        /*   a long value         */
#define LLOWD(x) ((UWORD)(x))
                                                        /* return high word of    */
                                                        /*   a long value         */
#define LHIWD(x) ((UWORD)(x >> 16))
                                                        /* return low byte of     */
                                                        /*   a word value         */
#define LLOBT(x) ((BYTE)(x & 0x00ff))
                                                        /* return high byte of    */
                                                        /*   a word value         */
#define LHIBT(x) ((BYTE)( (x >> 8) & 0x00ff))

/***************************************************************************/
```

```
#if I8086

                                        /* return long address   */
                                        /*   of short ptr        */
EXTERN LONG     ADDR();

                                        /* return long address   */
                                        /*   of the data seg     */
EXTERN LONG     LLDS();

                                        /* return long address   */
                                        /*   of the code seg     */
EXTERN LONG     LLCS();
                                        /* return a single byte  */
                                        /*   pointed at by long  */
                                        /*   ptr                 */
EXTERN BYTE     LBGET();
                                        /* set a single byte     */
                                        /*   pointed at by long  */
                                        /*   ptr, LBSET(lp, bt)  */
EXTERN BYTE     LBSET();
                                        /* return a single word  */
                                        /*   pointed at by long  */
                                        /*   ptr                 */
EXTERN WORD     LWGET();
                                        /* set a single word     */
                                        /*   pointed at by long  */
                                        /*   ptr, LWSET(lp, bt)  */
EXTERN WORD     LWSET();
                                        /* return a single long  */
                                        /*   pointed at by long  */
                                        /*   ptr                 */
EXTERN LONG     LLGET();
                                        /* set a single long     */
                                        /*   pointed at by long  */
                                        /*   ptr, LLSET(lp, bt)  */
EXTERN LONG     LLSET();
                                        /* return 0th byte of    */
                                        /*   a long value given  */
                                        /*   a short pointer to  */
                                        /*   the long value      */
#define LBYTE0(x) (*x)
                                        /* return 1st byte of    */
                                        /*   a long value given  */
                                        /*   a short pointer to  */
                                        /*   the long value      */
#define LBYTE1(x) (*(x+1))
                                        /* return 2nd byte of    */
                                        /*   a long value given  */
                                        /*   a short pointer to  */
                                        /*   the long value      */
#define LBYTE2(x) (*(x+2))
                                        /* return 3rd byte of    */
                                        /*   a long value given  */
                                        /*   a short pointer to  */
                                        /*   the long value      */
#define LBYTE3(x) (*(x+3))

#endif
```

```
/*************************************************************************/

#if MC68K
                                                /* return a long address*/
                                                /*  of a short pointer */
#define ADDR /**/

                                                /* return a single byte */
                                                /*   pointed at by long */
                                                /*   ptr                */
#define LBGET(x) ( (UBYTE) *((BYTE * )(x)) )
                                                /* set a single byte    */
                                                /*   pointed at by long */
                                                /*   ptr, LBSET(lp, bt) */
#define LBSET(x, y)  ( *((BYTE *)(x)) = y)
                                                /* return a single word */
                                                /*   pointed at by long */
                                                /*   ptr                */
#define LWGET(x) ( (WORD) *((WORD *)(x)) )
                                                /* set a single word    */
                                                /*   pointed at by long */
                                                /*   ptr, LWSET(lp, bt) */
#define LWSET(x, y)  ( *((WORD *)(x)) = y)

                                                /* return a single long */
                                                /*   pointed at by long */
                                                /*   ptr                */
#define LLGET(x) ( *((LONG *)(x)))
                                                /* set a single long    */
                                                /*   pointed at by long */
                                                /*   ptr, LLSET(lp, bt) */
#define LLSET(x, y) ( *((LONG *)(x)) = y)

                                                /* return 0th byte of   */
                                                /*   a long value given */
                                                /*   a short pointer to */
                                                /*   the long value     */
#define LBYTE0(x) ( *((x)+3) )
                                                /* return 1st byte of   */
                                                /*   a long value given */
                                                /*   a short pointer [illegible] */
                                                /*   the long value     */
#define LBYTE1(x) ( *((x)+2) )
                                                /* return 2nd byte of   */
                                                /*   a long value given */
                                                /*   a short pointer to */
                                                /*   the long value     */
#define LBYTE2(x) ( *((x)+1) )
                                                /* return 3rd byte of   */
                                                /*   a long value given */
                                                /*   a short pointer to */
                                                /*   the long value     */
#define LBYTE3(x) (*(x))


#endif
```

```
/*      TREEADDR.H      04/11/84 - 09/11/84     Gregg Morris            */

#define OB_NEXT(x) (tree + (x) * sizeof(OBJECT) + 0)
#define OB_HEAD(x) (tree + (x) * sizeof(OBJECT) + 2)
#define OB_TAIL(x) (tree + (x) * sizeof(OBJECT) + 4)
#define OB_TYPE(x) (tree + (x) * sizeof(OBJECT) + 6)
#define OB_FLAGS(x) (tree + (x) * sizeof(OBJECT) + 8)
#define OB_STATE(x) (tree + (x) * sizeof(OBJECT) + 10)
#define OB_SPEC(x) (tree + (x) * sizeof(OBJECT) + 12)
#define OB_X(x) (tree + (x) * sizeof(OBJECT) + 16)
#define OB_Y(x) (tree + (x) * sizeof(OBJECT) + 18)
#define OB_WIDTH(x) (tree + (x) * sizeof(OBJECT) + 20)
#define OB_HEIGHT(x) (tree + (x) * sizeof(OBJECT) + 22)


/*      OBDEFS.H        03/15/84 - 11/22/84     Gregg Morris            */

#define ROOT            0
#define NIL             -1
                                                /* max string length    */
#define MAX_LEN         81
                                                /* max depth of search  */
                                                /*    or draw for objects */
#define MAX_DEPTH       10
                                                /* inside patterns      */
#define IP_HOLLOW       0
#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
                                                /* system foreground and */
                                                /*    background rules  */
#define SYS_FG          0x1100

#define WTS_FG          0x11a1                  /* window title selected */
                                                /*    using pattern 2 & */
                                                /*    replace mode text */
#define WTN_FG          0x1100                  /* window title normal  */
                                                /* gsx modes            */
#define MD_REPLACE      1
#define MD_TRANS        2
#define MD_XOR          3
#define MD_ERASE        4
                                                /* gsx styles           */
#define FIS_HOLLOW      0
#define FIS_SOLID       1
#define FIS_PATTERN     2
#define FIS_HATCH       3
#define FIS_USER        4
                                                /* bit blt rules        */
#define ALL_WHITE       0
#define S_AND_D         1
#define S_ONLY          3
#define NOTS_AND_D      4
#define S_XOR_D         6
#define S_OR_D          7
#define D_INVERT        10
```

```
#define NOTS_OR_D       13
#define ALL_BLACK       15
                                                    /* font types        */
#define IBM             3
#define SMALL           5

/* Object Drawing Types */
                                                    /* Graphic types of obs */
#define G_BOX           20
#define G_TEXT          21
#define G_BOXTEXT       22
#define G_IMAGE         23
#define G_USERDEF       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
                                                    /* Object flags      */
#define NONE            0x0
#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
                                                    /* Object states     */
#define NORMAL          0x0
#define SELECTED        0x1
#define CROSSED         0x2
#define CHECKED         0x4
#define DISABLED        0x8
#define OUTLINED        0x10
#define SHADOWED        0x20
                                                    /* Object colors     */
#define WHITE           0
#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 OBJECT struct object
```

```
OBJECT
{
        WORD            ob_next;        /* -> object's next sibling        */
        WORD            ob_head;        /* -> head of object's children */
        WORD            ob_tail;        /* -> tail of object's children */
        UWORD           ob_type;        /* type of object- BOX, CHAR,...*/
        UWORD           ob_flags;       /* flags                           */
        UWORD           ob_state;       /* state- SELECTED, OPEN, ...      */
        LONG            ob_spec;        /* "out"- -> anything else         */
        UWORD           ob_x;           /* upper left corner of object     */
        UWORD           ob_y;           /* upper left corner of object     */
        UWORD           ob_width;       /* width of obj                    */
        UWORD           ob_height;      /* height of obj                   */
};

#define ORECT   struct orect

ORECT
{
        ORECT   *o_link;
        WORD    o_x;
        WORD    o_y;
        WORD    o_w;
        WORD    o_h;
} ;

#define GRECT   struct grect

GRECT
{
        WORD    g_x;
        WORD    g_y;
        WORD    g_w;
        WORD    g_h;
} ;

#define TEDINFO struct text_edinfo

TEDINFO
{
        LONG            te_ptext;       /* ptr to text                     */
        LONG            te_ptmplt;      /* ptr to template                 */
        LONG            te_pvalid;      /* ptr to validation chrs.         */
        WORD            te_font;        /* font                            */
        WORD            te_junk1;       /* junk word                       */
        WORD            te_just;        /* justification- left, right...*/
        WORD            te_color;       /* color information word          */
        WORD            te_junk2;       /* junk word                       */
        WORD            te_thickness;   /* border thickness                */
        WORD            te_txtlen;      /* length of text string           */
        WORD            te_tmplen;      /* length of template string       */
};

#define ICONBLK struct icon_block

ICONBLK
{
        LONG    ib_pmask;
        LONG    ib_pdata;
        LONG    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;
};

#define BITBLK struct bit_block

BITBLK
{
        LONG    bi_pdata;               /* ptr to bit forms data        */
        WORD    bi_wb;                  /* width of form in bytes       */
        WORD    bi_hl;                  /* height in lines              */
        WORD    bi_x;                   /* source x in bit form         */
        WORD    bi_y;                   /* source y in bit form         */
        WORD    bi_rule;                /* blt rule to use              */
};

#define USERBLK struct user_blk
USERBLK
{
        LONG    ub_code;
        LONG    ub_parm;
};

#define PARMBLK struct parm_blk
PARMBLK
{
        LONG    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;
};

#define MFDB struct memform
MFDB
{
        LONG        mp;
        WORD        fwp;
        WORD        fh;
        WORD        fww;
        WORD        ff;
        WORD        np;
        WORD        r1;
        WORD        r2;
        WORD        r3;
    };
```

```
#define FILLPAT struct patarray
FILLPAT
{
    WORD        patword[16];
};

#define EDSTART         0
#define EDINIT          1
#define EDCHAR          2
#define EDEND           3

#define TE_LEFT         0
#define TE_RIGHT        1
#define TE_CNTR         2
/*
                        DEMO.H
*/
#define DEMOMENU        0       /* TREE */
#define DEMOINFO        8       /* OBJECT in TREE #0 */
#define DEMODESK        3       /* OBJECT in TREE #0 */
#define DEMOFILE        4       /* OBJECT in TREE #0 */
#define DEMOOPTS        5       /* OBJECT in TREE #0 */
#define DEMOLOAD        17      /* OBJECT in TREE #0 */
#define DEMOSAVE        18      /* OBJECT in TREE #0 */
#define DEMOSVAS        19      /* OBJECT in TREE #0 */
#define DEMOABAN        20      /* OBJECT in TREE #0 */
#define DEMOQUIT        21      /* OBJECT in TREE #0 */
#define DEMOPENS        23      /* OBJECT in TREE #0 */
#define DEMOERAP        25      /* OBJECT in TREE #0 */
#define DEMOINFD        1       /* TREE */
#define DEMOOK          6       /* OBJECT in TREE #1 */
#define DEMOPEND        3       /* TREE */
#define DEMOPSOK        10      /* OBJECT in TREE #3 */
#define DEMOCNCL        12      /* OBJECT in TREE #3 */
#define DEMOOVWR        0       /* STRING */
#define DEMONWDW        1       /* STRING */
#define DEMOSVAD        2       /* TREE */
#define DEMOSOK         2       /* OBJECT in TREE #2 */
#define DEMOSCNL        3       /* OBJECT in TREE #2 */
#define DEMONAME        4       /* OBJECT in TREE #2 */
#define DEMOIMG         3       /* OBJECT in TREE #1 */
#define DEMOPCLR        18      /* OBJECT in TREE #3 */
#define DEMOPFIN        3       /* OBJECT in TREE #3 */
#define DEMOPMED        4       /* OBJECT in TREE #3 */
#define DEMOPBRD        5       /* OBJECT in TREE #3 */
#define DEMOEFIN        6       /* OBJECT in TREE #3 */
#define DEMOEMED        7       /* OBJECT in TREE #3 */
#define DEMOEBRD        8       /* OBJECT in TREE #3 */

/*      GEMBIND.H       05/05/84 - 08/13/85     Lee Lorenzen            */

                                /* 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_BVSET      16
#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
#define MENU_UNREGISTER 36
                                  /* 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
                                   /* GEM function 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[8]
#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 AP_BVDISK          int_in[0]
#define AP_BVHARD          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 MMO1_FLAGS          int_in[4]
#define MMO1_X              int_in[5]
#define MMO1_Y              int_in[6]
#define MMO1_WIDTH          int_in[7]
#define MMO1_HEIGHT         int_in[8]

#define MMO2_FLAGS          int_in[9]
#define MMO2_X              int_in[10]
#define MMO2_Y              int_in[11]
#define MMO2_WIDTH          int_in[12]
#define MMO2_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 MM_MID          int_in[0]       /* unregister       */
#define CHECK_IT        int_in[1]       /* icheck           */
#define ENABLE_IT       int_in[1]       /* ienable          */

#define TITLE_NUM       int_in[0]       /* tnorm            */
#define NORMAL_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

#define FMD_FORWARD     0
#define FMD_BACKWARD    1
#define FMD_DEFLT       2

#define FM_OBJ          int_in[0]
#define FM_ICHAR        int_in[1]
#define FM_INXTOB       int_in[2]

#define FM_ONXTOB       int_out[1]
#define FM_OCHAR        int_out[2]

#define FM_CLKS         int_in[1]
                                        /* 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_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
#define WF_SCREEN       17
#define WF_TATTRB       18
                                        /* 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 parameters       */
#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]
                                 /*    End GEMBIND.H    */
```

```
/*************************************************************************/
/*      File:   demo.c  Re-ordered functions & cleaned code              */
/*************************************************************************/
/*                                                                       */
/*                GGGGG          EEEEEEEE       MM      MM               */
/*              GG               EE             MMMM  MMMM               */
/*              GG   GGG         EEEEE          MM  MM  MM               */
/*              GG    GG         EE             MM      MM               */
/*                GGGGG          EEEEEEEE       MM      MM               */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*                          +-----------------------+                    */
/*                          | Digital Research, Inc. |                   */
/*                          | 60 Garden Court        |                   */
/*                          | Monterey, CA.    93940 |                   */
/*                          +-----------------------+                    */
/*                                                                       */
/*   The  source code  contained in  this listing is a non-copyrighted   */
/*   work which  can be  freely used.  In applications of  this source   */
/*   code you  are requested to  acknowledge Digital Research, Inc. as   */
/*   the originator of this code.                                        */
/*                                                                       */
/*   Author:    Tom Rolander, Tim Oren                                   */
/*   PRODUCT:   GEM Sample Application                                   */
/*   Module:    DEMO,  Version 1.1                                       */
/*   Version:   March 22, 1985                                           */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*               Modified by Phillip Balma and Bill Fitler               */
/*                            February, 1986                             */
/*

Page*/
/*-------------------------------*/
/*      includes                 */
/*-------------------------------*/

#include "portab.h"                             /* portable coding conv */
#include "machine.h"                            /* machine depndnt conv */
#include "obdefs.h"                             /* object definitions   */
#include "treeaddr.h"                           /* tree address macros  */
#include "gembind.h"                            /* gem binding structs  */
#include "demo.h"                               /* demo resources       */
/**/                                            /*    file offsets      */


/*-------------------------------*/
/*      defines                  */
/*-------------------------------*/

#define ARROW           0
#define HOUR_GLASS      2

#define DESK            0

#define END_UPDATE      0
#define BEG_UPDATE      1
```

```
#define PEN_INK         BLACK
#define PEN_ERASER      WHITE

#define PEN_FINE        1
#define PEN_MEDIUM      5
#define PEN_BROAD       9

#define X_FWD           0x0100          /* extended object types */
#define X_BAK           0x0200          /* used with scrolling   */
#define X_SEL           0x0300          /* selectors             */
#define N_COLORS        15L

#define YSCALE(x) UMUL_DIV(x, scrn_xsize, scrn_ysize)

#define TE_TXTLEN(x) (x + 24)
#define BI_PDATA(x)     (x)
#define BI_WB(x)        (x + 4)
#define BI_HL(x)        (x + 6)


/*----------------------------*/
/*      External Functions    */
/*----------------------------*/

EXTERN LONG     dos_alloc();


/*


Page*/
/***********************************************************************/
/***********************************************************************/
/****                                                              ****/
/****                       Data Structures                        ****/
/****                                                              ****/
/***********************************************************************/
/***********************************************************************/


/*----------------------------*/
/*  External Data Structures  */
/*----------------------------*/

EXTERN  UWORD   DOS_ERR;
EXTERN  LONG    drawaddr;

/*----------------------------*/
/*     Global Data Structures */
/*----------------------------*/

GLOBAL WORD     contrl[11];             /* control inputs               */
GLOBAL WORD     intin[80];              /* max string length            */
GLOBAL WORD     ptsin[256];             /* polygon fill points          */
GLOBAL WORD     intout[45];             /* open workstation output      */
GLOBAL WORD     ptsout[12];


/*----------------------------*/
/*     Local Data Structures  */
/*----------------------------*/
```

```
WORD     char_width;                        /* character width              */
WORD     char_hite;                         /* character height             */
WORD     box_width;                         /* box (cell) width             */
WORD     box_hite;                          /* box (cell) height            */
WORD     space_hite;                        /* height of space between lines*/
WORD     gem_handle;                        /* GEM vdi handle               */
WORD     vdi_handle;                        /* demo vdi handle              */
WORD     work_out[57];                      /* open virt workstation values */
GRECT    scrn_area;                         /* whole screen area            */
GRECT    work_area;                         /* drawing area of main window  */
GRECT    draw_area;                         /* area equal to work area      */
GRECT    save_area;                         /* save area for full/unfulling */
WORD     msg_buff[8];                       /* message buffer               */
LONG     addr_msg;                          /* LONG pointer to message bfr  */
LONG     addr_menu;                         /* menu tree address            */
WORD     full_x;                            /* full window 'x'              */
WORD     full_y;                            /* full window 'y'              */
WORD     full_width;                        /* full window 'w'              */
WORD     full_hite;                         /* full window 'h'              */
WORD     scrn_xsize;                        /* width of one pixel           */
WORD     scrn_ysize;                        /* height of one pixel          */
UWORD    m_out              = FALSE;        /* mouse in/out of window flag  */
WORD     ev_which;                          /* event multi return state(s)  */
UWORD    mousex, mousey;                    /* mouse x,y position           */
UWORD    bstate, bclicks;                   /* button state, & # of clicks  */
UWORD    kstate, kreturn;                   /* key state and keyboard char  */
MFDB     draw_mfdb;                         /* draw buffer mmry frm def blk */
MFDB     scrn_mfdb;                         /* screen memory form defn blk  */
LONG     buff_size;                         /* buffer size req'd for screen */
LONG     buff_location;                     /* screen buffer pointer        */
WORD     demo_whndl;                        /* demo window handle           */
WORD     demo_shade         = PEN_INK;      /* demo current pen shade       */
WORD     pen_shade          = PEN_INK;      /* saved pen shade              */
WORD     demo_pen           = 1;            /* demo current pen width       */
WORD     demo_height        = 4;            /* demo current char height     */
WORD     char_fine;                         /* character height for fine    */
WORD     char_medium;                       /* character height for medium  */
WORD     char_broad;                        /* character height for broad   */
WORD     monumber           = 5;            /* mouse form number            */
LONG     mofaddr            = 0x0L;         /* mouse form address           */
WORD     file_handle;                       /* file handle -> pict ld/sv    */
BYTE     file_name[64]      = "";           /* current pict file name       */
BOOLEAN key_input;                          /* key inputting state          */
WORD     key_xbeg;                          /* x position for line beginning*/
WORD     key_ybeg;                          /* y position for line beginning*/
WORD     key_xcurr;                         /* current x position           */
WORD     key_ycurr;                         /* current y position           */
                                            /* demo window title            */
BYTE     *wdw_title         = " New DEMO ";

WORD     usercolor[2]       = {1, 0};
MFDB     userbrush_mfdb;
USERBLK brushub[6];
LONG     color_sel[N_COLORS+1] = {           /* data for scrolling color selector *

                            N_COLORS,
                            0x31FF1071L,
                            0x32FF1072L,
                            0x33FF1073L,
                            0x34FF1074L,
                            0x35FF1075L,
                            0x36FF1076L,
                            0x37FF1077L,
                                0x38FF1078L,
                                0x39FF1079L,
                                0x41FF107AL,
                                0x42FF107BL,
                                0x43FF107CL,
                                0x44FF107DL,
                                0x45FF107EL,
                                0x46FF107FL};

/*------------------------------*/
/*      Mouse Data Structures   */
/*------------------------------*/

WORD    erase_broad[37] =               /* mouse form for broad eraser  */
{
        7, 7, 1, 0, 1,
        0x0000, 0x0000, 0x0000, 0x0000, /* mask */
        0x0000, 0x1ff0, 0x1ff0, 0x1ff0,
        0x1ff0, 0x1ff0, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, /* data */
        0x7ffc, 0x600c, 0x600c, 0x600c,
        0x600c, 0x600c, 0x7ffc, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000
};
WORD    erase_medium[37] =              /* mouse form for medium eraser */
{
        7, 7, 1, 0, 1,
        0x0000, 0x0000, 0x0000, 0x0000, /* mask */
        0x0000, 0x0000, 0x07c0, 0x07c0,
        0x07c0, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, /* data */
        0x0000, 0x1ff0, 0x1830, 0x1830,
        0x1830, 0x1ff0, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000
};
WORD    erase_fine[37] =                /* mouse form for fine eraser   */
{
        7, 7, 1, 0, 1,
        0x0000, 0x0000, 0x0000, 0x0000, /* mask */
        0x0000, 0x0000, 0x0000, 0x0100,
        0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000, /* data */
        0x0000, 0x0000, 0x07c0, 0x06c0,
        0x07c0, 0x0000, 0x0000, 0x0000,
        0x0000, 0x0000, 0x0000, 0x0000
};

LONG    string_addr();

/*------------------------------*/
/*      GEMAIN                  */
/*------------------------------*/
GEMAIN()
{
        WORD    term_type;

        if (!(term_type = demo_init()))  demo();
        demo_term(term_type);
}


/*------------------------------*/
/*      add file name           */
/*------------------------------*/
VOID
add_file_name(dname, fname)         /* replace name at end of input file spec*/
BYTE    *dname, *fname;
{
        BYTE    c;
        WORD    ii;


        ii = strlen(dname);
        while (ii && (((c = dname[ii-1])  != '\\') && (c != ':')))
                ii--;
        dname[ii] = '\0';
        strcat(dname, fname);
}


/*------------------------------*/
/*      align_x                 */
/*------------------------------*/
WORD
align_x(x)              /* forces word alignment for column position    */
WORD    x;              /*      rounding to nearest word                */
{
        return((x & 0xfff0) + ((x & 0x000c) ? 0x0010 : 0));
}


/*------------------------------*/
/*      cursor                  */
/*------------------------------*/
VOID
cursor(color)                   /* turn cursor on,  color = BLACK       */
WORD    color;                  /*      or cursor off, color = WHITE    */
{
        WORD    pxy[4];

        pxy[0] = key_xcurr + 1;
        pxy[1] = key_ycurr + space_hite;
        pxy[2] = key_xcurr + 1;
        pxy[3] = key_ycurr - box_hite;

        vsl_color(vdi_handle,color);
        vswr_mode(vdi_handle,MD_REPLACE);
        vsl_type (vdi_handle,FIS_SOLID);
        vsl_width (vdi_handle,PEN_FINE);
        graf_mouse(M_OFF, 0x0L);
        v_pline(vdi_handle, 2, pxy);
        graf_mouse(M_ON, 0x0L);
}
```

```
/*------------------------------*/
/*      curs on                 */
/*------------------------------*/
VOID    curs_on()               /* Turn 'soft' cursor 'on'      */
{
        cursor(pen_shade);
}

/*------------------------------*/
/*      curs_off                */
/*------------------------------*/
VOID    curs_off()              /* Turn 'soft' cursor 'off'     */
{
        cursor(PEN_ERASER);
}

/*------------------------------*/
/*      demo                    */
/*------------------------------*/
demo()
{
        WORD    junk;           /* only used as a placeholder for the AES
                                   bindings.  It is used in the evnt_multi
                                   call in place of an address of a
                                   keyboard handler, because such addresses
                                   cannot be a NULLPTR!!!  */

        FOREVER
        {
                                /* Wait for mouse, message events only,
                                   left button goes down, mouse
                                   exits rectangle.
                                   addr_msg = address of message buffer  */

                ev_which = evnt_multi(MU_BUTTON | MU_MESAG | MU_M1,
                                0x02, 0x01, 0x01,
                                m_out,
                                (UWORD) work_area.g_x, (UWORD) work_area.g_y,
                                (UWORD) work_area.g_w, (UWORD) work_area.g_h,
                                0, 0, 0, 0, 0,
                                addr_msg, 0, 0,
                                &mousex, &mousey, &bstate, &kstate,
                                &junk, &bclicks);

                wind_update(BEG_UPDATE);

                                /* If hndl_msg returns TRUE then the user
                                   either clicked on the close button, or
                                   selected the QUIT option in the FILE menu *


                if (ev_which & MU_MESAG) if (hndl_msg())  break;

                if (ev_which & MU_BUTTON) hndl_button();

                if (ev_which & MU_M1) hndl_mouse();

                wind_update(END_UPDATE);
        }
```

```
}


/*--------------------------------------------------------------------------*/
/*      demo_init
----------------------------------------------------------------------------*/

WORD demo_init() {

        WORD    work_in[11];
        WORD    i;
                                /* Initialize libraries, if error, bail out */

        if (appl_init() == -1) return(4);

                                /* Lock screen to begin update */
        wind_update(BEG_UPDATE);

                                /* Change mouse form to hour glass */

        graf_mouse(HOUR_GLASS, 0x0L);

                                /* Load the Resource Construction Set
                                   information contained in DEMO.RSC;
                                   Menu bar, titles, info, etc. */

        if (!rsrc_load( ADDR("DEMO.RSC"))){

                                /* Error trying to load file.
                                   Restore mouse form */

        graf_mouse(ARROW, 0x0L);
        form_alert(1,
            ADDR("[3][Fatal Error !|DEMO.RSC|File Not Found][ Abort ]"));
        return(1);
        }
                                /* Get the physical device handle,
                                   character cell size, and size
                                   of box that holds cell of
                                   system font.*/

        vdi_handle = graf_handle(&char_width,&char_hite,&box_width,&box_hite);

                                 /* Set virtual workstation values
                                    to defaults */
        for (i=0; i<10; i++) {
                work_in[i]=1;
        }
        work_in[10]=2;          /* Use RC coordinates */
                                /* Open virtual work station.
                                   Changes the physical handle to a
                                   virtual handle.  */

        v_opnvwk(work_in,&vdi_handle,work_out);

                                /* Bail out if device cannot be opened */

        if (vdi_handle == 0) return(1);

                                /* Save work values in local temps before
```

```
                                  making next VDI call and blowing them
                                  away.  */

   draw_mfdb.fwp = work_out[0] + 1;/* screen width in pixels */
   draw_mfdb.fh = work_out[1] + 1; /* screen height in pixels */
   scrn_xsize = work_out[3];       /* width (raster) aspect ratio */
   scrn_ysize = work_out[4];       /* height (raster) aspect ratio */
   char_fine = work_out[46];       /* min char height ptsout(1) */
   char_medium = work_out[48];     /* max char height ptsout(3) */
   char_broad = char_medium * 2;

                               /* Get number of planes */

   vq_extnd(vdi_handle, 1, work_out);
   draw_mfdb.np = work_out[4];     /* number of planes */

                               /* screen width in words */

   draw_mfdb.fww = draw_mfdb.fwp>>4;
   draw_mfdb.ff = 0;

                               /* Compute size of screen buffer (bytes):
                                  width (pixels)/8 * height * planes */

   buff_size = (LONG)(draw_mfdb.fwp>>3) *
               (LONG)draw_mfdb.fh *
               (LONG)draw_mfdb.np;

                               /* Allocate a screen buffer */

   buff_location = draw_mfdb.mp  = dos_alloc(buff_size);

                               /* Enough Contiguous Memory?*/

   if (draw_mfdb.mp == 0) return(2);

   scrn_area.g_x = 0;
   scrn_area.g_y = 0;
   scrn_area.g_w = draw_mfdb.fwp;
   scrn_area.g_h = draw_mfdb.fh;

                               /* Set to physical screen, so ignore
                                  other parameters on vro_cpyfm */

   scrn_mfdb.mp = 0x0L;

                               /* Clear the screen buffer so that
                                  demo displays a clean initial
                                  screen. First copy the GRECTs
                                  and then do a bit by bit copy. */

   rc_copy(&scrn_area, &draw_area);

                               /* Now clear all the bits in draw.
                                  The source is the scrn area, and the
                                  destination is draw.  The logical
                                  operation is clear the destination bits,
                                  and because there is a 0 in the pointer
                                  field of the scrn_mfdb, all the other mfdb
                                  parameters are ignored, and the source is
                                  is treated as the physical device.  */
```

```
rast_op(0,&scrn_area,&scrn_mfdb,&draw_area,&draw_mfdb);

                        /* Transforms user defined objects */
pict_init();

                        /* Address of 16 byte message buffer */

addr_msg = ADDR((BYTE *) &msg_buff[0]);

                        /* Get the size of the DESKTOP
                           window.  Make DEMO window same size.  */

wind_get(DESK, WF_WXYWH, &full_x, &full_y, &full_width, &full_hite);

                        /* Get the addr of the menu tree in the object
                           tree for DEMO.  This tree is loaded in
                           memory by the rsrc_load call at the
                           beginning of this routine.  */

rsrc_gaddr(R_TREE, DEMOMENU, &addr_menu);

                        /* Show the menu for DEMO */

menu_bar(addr_menu, TRUE);

                        /* Create DEMO window with following:

                            FEF means
                           F00 - vertical and horizontal
                                 slider, left and right arrows

                            E0  - Up and down arrows, size box

                            F   - Move, full, and close
                                  boxes, and title bar */

demo_whndl = wind_create(0x0fef, full_x - 1, full_y,
                         full_width, full_hite);
if (demo_whndl == -1)
{                       /* No more windows available */

        form_alert(1, string_addr(DEMONWDW));
        return(3);
}
                        /* Enter name of window.
                           Last 2 parameters (0,0) must
                           be there because of syntax. */

wind_set(demo_whndl, WF_NAME, ADDR(wdw_title), 0, 0);

full_x = align_x(full_x);

                        /* Display an expanding box.  The initial
                           values are arbitrary. */

graf_growbox(full_width/2, full_hite/2, 21, 21,
             full_x, full_y, full_width, full_hite);

                        /* Finally open and display the demo window */

wind_open(demo_whndl,full_x,full_y,full_width,full_hite);
```

```
                                        /* set_work will get the actual size of
                                           the window work area, and update the
                                           draw area. The TRUE refers to whether the
                                           slider area needs to be updated.  The
                                           only time set_work is not called to
                                           update the slider area is when the window
                                           has been moved (doesn't change slider's
                                           relative position within the window). */
        set_work(TRUE);
                                        /* Save current draw area in previous
                                           variable.  Used in do_full.  */

        rc_copy (&draw_area, &save_area);

        graf_mouse(ARROW,0x0L); /* Restore arrow cursor */
        wind_update(END_UPDATE);/* Unlock update region */
        return(0);
}                                       /* end demo_init */


/*------------------------------*/
/*      demo_term               */
/*------------------------------*/
demo_term(term_type)
WORD    term_type;
{
        WORD x, y, w, h;

        switch (term_type)      /* NOTE: all cases fall through         */
        {
            case (0):           /* Normal termination.
                                   Get the current GRECT, close the
                                   window, display a shrinking box, and then
                                   free all AES window resources.  */

                wind_get(demo_whndl, WF_CXYWH, &x, &y, &w, &h);
                wind_close(demo_whndl);
                graf_shrinkbox(full_width/2, full_hite/2, 21, 21,
                                x, y, w, h);
                wind_delete(demo_whndl);

            case (3):           /* No more windows available.  We have loaded
                                   menu and allocated memory for screen. */

                menu_bar(0x0L, FALSE);
                dos_free(draw_mfdb.mp);

            case (2):           /* Couldn't open device. */

                v_clsvwk( vdi_handle );

            case (1):           /* Couldn't find RSC file. */

                wind_update(END_UPDATE);
                appl_exit();

            case (4):           /* Error on appl_init(). */

                break;
        }
}
```

```
/*------------------------------*/
/*      desel_obj               */
/*------------------------------*/
VOID
desel_obj(tree, which)          /* Turn off selected bit of specfied object */
LONG    tree;
WORD    which;
{
        undo_obj(tree, which, SELECTED);
}


/*------------------------------*/
/*      dial_name               */
/*------------------------------*/
WORD    dial_name ( name )                /* dialogue box input filename  */
BYTE    *name;
{
        LONG    tree ;
        LONG    ted_addr ;
        BYTE    c ;
        WORD    i, j;
        GRECT   box;


        objc_xywh(addr_menu, DEMOFILE, &box);
        rsrc_gaddr( R_TREE, DEMOSVAD, &tree) ;
        ted_addr = LLGET(OB_SPEC(DEMONAME));
        LLSET( ted_addr, ADDR(name) ) ;
        LWSET( TE_TXTLEN(ted_addr),8);
        name[0] = '\0';
        if (hndl_dial(tree, DEMONAME, box.g_x, box.g_y,
                      box.g_w, box.g_h) == DEMOSOK)
        {
                i = j =  0;
                while (TRUE)
                {
                        c = name[i++];
                        if (!c) break ;
                        if ( (c != ' ') && (c != '_') ) name[j++] = c ;
                }
                if ( *name ) strcpy( &name[j], ".DOO" ) ;
                desel_obj(tree, DEMOSOK);
                return (TRUE);
        }
        else
        {
                desel_obj(tree, DEMOSCNL);
                return (FALSE);
        }
}

/*------------------------------*/
/*      dir_obj                 */
/*------------------------------*/
VOID
dir_obj(tree, which)
LONG    tree;
WORD    which;
{
        unflag_obj(tree, which, INDIRECT);
}



/*------------------------------*/
/*      do_full                 */
/*------------------------------*/
VOID
do_full(wh)     /* Depending on current window state, either make window*/
WORD    wh;     /*   full size -or- return to previous shrunken size    */

{
        GRECT   prev;
        GRECT   curr;
        GRECT   full;

        graf_mouse(M_OFF,0x0L);
                                /* Get full & current screen size */

        wind_get(wh, WF_FXYWH, &full.g_x, &full.g_y, &full.g_w, &full.g_h);
        wind_get(wh, WF_CXYWH, &curr.g_x, &curr.g_y, &curr.g_w, &curr.g_h);

        if ( rc_equal(&curr, &full) )
        {                       /* Full now, so change to previous size,
                                   setting current size to previous.*/

                wind_get(wh, WF_PXYWH, &prev.g_x, &prev.g_y,
                        &prev.g_w, &prev.g_h);

                                /* If previous = full = current, then do
                                   nothing.  */

                if (!( rc_equal(&prev, &full) )) {

                        graf_shrinkbox(prev.g_x, prev.g_y, prev.g_w, prev.g_h,
                                full.g_x, full.g_y, full.g_w, full.g_h)

                        wind_set(wh, WF_CXYWH, prev.g_x, prev.g_y,
                                prev.g_w, prev.g_h);

                        rc_copy(&save_area, &draw_area);
                        set_work(TRUE);
                }
        }
        else
        {                       /* Not full, so make it full. Set the
                                   max window size as the full size.

                                   Save the current area , in case we toggle
                                   between full and non-full sizes */

                rc_copy(&draw_area, &save_area);
                graf_growbox(curr.g_x, curr.g_y, curr.g_w, curr.g_h,
                        full.g_x, full.g_y, full.g_w, full.g_h);

                wind_set(wh, WF_CXYWH, full.g_x, full.g_y, full.g_w, full.g_h)
                set_work(TRUE);
        }
        graf_mouse(M_ON,0x0L);
```

```
}


/*------------------------------*/
/*      do_open                 */
/*------------------------------*/
VOID    do_open(wh, org_x, org_y, x, y, w, h)
                                /* grow and open specified window */
WORD    wh;
WORD    org_x, org_y;
WORD    x, y, w, h;
{
        graf_growbox(org_x, org_y, 21, 21, x, y, w, h);
        wind_open(wh, x, y, w, h);
}

/*------------------------------*/
/*      do_load                 */
/*------------------------------*/
VOID    do_load(need_name)              /* load demo picture file       */
BOOLEAN need_name;
{
        if (!need_name || get_file(TRUE))
        {
                if (!DOS_ERR)
                {
                        dos_read(file_handle, buff_size,buff_location);
                        dos_close(file_handle);
                        enab_menu(DEMOSAVE);
                        enab_menu(DEMOABAN);
                        restore_work();
                }
        }
}


/*------------------------------*/
/*      do_penselect            */
/*------------------------------*/
VOID    do_penselect()                  /* use dialog box to input selection    */
{                                       /*   of specified pen/eraser            */

        WORD    exit_obj, psel_obj, color;
        LONG    tree, bind[2];
        GRECT   box;

                                        /* Get the coordinates of the pen/eraser
                                           selection item in OPTIONS menu. Only used
                                           to display a growing/shrinking box  */

        objc_xywh(addr_menu, DEMOPENS, &box);

                                        /* Get the address of the pencil/eraser
                                           dialog that has already been loaded in
                                           memory (see demo_init)  */

        rsrc_gaddr(R_TREE, DEMOPEND, &tree);

                                        /* First determine and set the current
```

```
                                  selection state */

switch (demo_pen) {
        case PEN_FINE:
         sel_obj(tree, (demo_shade != PEN_ERASER)? DEMOPFIN: DEMOEFIN)
         break;
        case PEN_MEDIUM:
         sel_obj(tree, (demo_shade != PEN_ERASER)? DEMOPMED: DEMOEMED)
         break;
        case PEN_BROAD:
         sel_obj(tree, (demo_shade != PEN_ERASER)? DEMOPBRD: DEMOEBRD)
         break;
}

                              /* Put the address of the color tables in
                                 the object spec of G_IBOX parent of the
                                 color selection panel */

set_select(tree, DEMOPCLR, pen_shade - 1, bind, color_sel);


                              /* Display the dialog and get the user
                                 selection (exitobj)  */

exit_obj = hndl_dial(tree, 0, box.g_x, box.g_y, box.g_w, box.g_h);

                              /* Clear the selected object item. Note
                                 that  when the loop is exited, we are
                                 pointing at the last selected object */

for (psel_obj = DEMOPFIN; psel_obj <= DEMOEBRD; psel_obj++)
        if (LWGET(OB_STATE(psel_obj)) & SELECTED)
        {
                desel_obj(tree, psel_obj);
                break;
        }

color = get_select(tree, DEMOPCLR) + 1;

if (exit_obj == DEMOPSOK)
{
        switch (psel_obj) {
                case DEMOPFIN:
                        set_pen(PEN_FINE, char_fine);
                        demo_shade = color;
                        break;
                case DEMOPMED:
                        set_pen(PEN_MEDIUM, char_medium);
                        demo_shade = color;
                        break;
                case DEMOPBRD:
                        set_pen(PEN_BROAD, char_broad);
                        demo_shade = color;
                        break;
                case DEMOEFIN:
                        set_eraser(PEN_FINE, char_fine,
                                (BYTE *) erase_fine);
                        break;
                case DEMOEMED:
                        set_eraser(PEN_MEDIUM, char_medium,
                                (BYTE *) erase_medium);
```

```
                                        break;
                                case DEMOEBRD:
                                        set_eraser(PEN_BROAD, char_broad,
                                                (BYTE *) erase_broad);
                                        break;
                        }
                        pen_shade = color;
                        desel_obj(tree, DEMOPSOK);
                }
                else
                        desel_obj(tree, DEMOCNCL);
}

/*------------------------------*/
/*      do_redraw               */
/*------------------------------*/
VOID
do_redraw(wh, area)             /* Redraw specified area from draw buffer */
WORD    wh;
GRECT   *area;
{
        GRECT   box, dirty_source;

        graf_mouse(M_OFF, 0x0L);

        wind_get(wh, WF_FIRSTXYWH, &box.g_x, &box.g_y, &box.g_w, &box.g_h);
        while ( box.g_w && box.g_h ){
          if (rc_intersect(area, &box)){
                if (rc_intersect(&work_area, &box)){
                        dirty_source.g_x = (box.g_x - work_area.g_x) +
                                        draw_area.g_x;
                        dirty_source.g_y = (box.g_y - work_area.g_y) +
                                        draw_area.g_y;
                        dirty_source.g_w = box.g_w;
                        dirty_source.g_h = box.g_h;

                        rast_op(3, &dirty_source, &draw_mfdb,
                                &box, &scrn_mfdb);
                }
          }
          wind_get(wh, WF_NEXTXYWH, &box.g_x, &box.g_y, &box.g_w, &box.g_h);
        }               /* end while more rectangles... */
        graf_mouse(M_ON, 0x0L);
}

/*------------------------------*/
/*      do_save                 */
/*------------------------------*/
VOID    do_save()               /* Save current named DEMO picture */
{
        if (*file_name)
        {
                file_handle = dos_open(ADDR(file_name),2);
                if (DOS_ERR) file_handle = dos_create(ADDR(file_name),0);
                else
                    if (form_alert(1, string_addr(DEMOOVWR)) == 2) return;

                dos_write(file_handle, buff_size, buff_location);
                enab_menu(DEMOSAVE);
                enab_menu(DEMOABAN);
                dos_close(file_handle);
        }
```

```
}

/*------------------------------*/
/*      do_save_as              */
/*------------------------------*/
VOID    do_svas()               /* Save DEMO picture as named   */
{
        BYTE    name[13];

        if (dial_name(&name[0]))
        {
                add_file_name(file_name, name);
                do_save();
        }
}


/*------------------------------*/
/*      draw_pencil             */
/*------------------------------*/
WORD    draw_pencil(x, y)
UWORD   x, y;
{
                                /* Left button is down. Trace lines */
        UWORD   pxy[4];
        WORD    done;
        UWORD   mflags;
        UWORD   locount, hicount;
        UWORD   ev_which, bbutton, kstate, kreturn, breturn;

                                /* Clip all lines at borders of work area.
                                If this is not set here, then user can draw
                                over the border areas of the window.  */

        set_clip(TRUE, &work_area);
        pxy[0] = x;             /* Save old mouse position */
        pxy[1] = y;
                                /* Set color to current shade (could be
                                eraser), replace writing mode and solid
                                line style.  */

        vsl_color(vdi_handle,demo_shade);
        vswr_mode(vdi_handle,MD_REPLACE);
        vsl_type (vdi_handle,FIS_SOLID);

        if (demo_shade != PEN_ERASER)
        {
                vsl_width (vdi_handle,demo_pen);

                                /* Set for rounded ends style */

                vsl_ends(vdi_handle, 2, 2);
                hicount = 0;
                locount = 125;
                mflags = MU_BUTTON | MU_M1 | MU_TIMER;
                graf_mouse(M_OFF, 0x0L);
        }
        else {
                                /* Set fill style to solid */
                vsf_interior(vdi_handle, 1);
                                        /* Set color to white */

                vsf_color(vdi_handle, WHITE);
                mflags = MU_BUTTON | MU_M1;
        }

        done = FALSE;
        while (!done)
        {
                                        /* Wait for mouse to
                                                1. Leave the rectangle
                                                   at (pxy[0], pxy[1]),
                                                   no double clicks,
                                                2. Left button up (button is down),
                                                3. Wait for timer to expire
                                                   (125 millisecs). */

                ev_which = evnt_multi(mflags,
                                        0x01, 0x01, 0x00,
                                        1, pxy[0], pxy[1], 1, 1,
                                        0, 0, 0, 0, 0,
                                        addr_msg, locount, hicount,
                                        &pxy[2], &pxy[3], &bbutton, &kstate,
                                        &kreturn, &breturn);

                if (ev_which & MU_BUTTON)
                {
                                        /* Left button up */

                        if (!(mflags & MU_TIMER)) graf_mouse(M_OFF, 0x0L);

                                        /* Connect the dots(old & new mouse pos.) */

                        if (demo_shade != PEN_ERASER)
                                v_pline(vdi_handle, 2, (WORD *) pxy);
                        else
                                eraser((WORD) pxy[2], (WORD) pxy[3]);

                        graf_mouse(M_ON, 0x0L);
                        done = TRUE;
                }
                else
                        if (ev_which & MU_TIMER)
                        {
                                /* Timer expired, no mouse movement,
                                   try again. */

                                graf_mouse(M_ON, 0x0L);
                                mflags = MU_BUTTON | MU_M1;
                        }
                        else {
                                /* Mouse moved , but left button still
                                   down, so continue tracking mouse. */

                                if (!(mflags & MU_TIMER))
                                        graf_mouse(M_OFF, 0x0L);

                                if (demo_shade != PEN_ERASER)
                        {
                                v_pline(vdi_handle, 2, (WORD *) pxy);
                                mflags = MU_BUTTON | MU_M1 | MU_TIMER;
                        }
                        else {
                                eraser((WORD) pxy[2], (WORD) pxy[3]);
                                graf_mouse(M_ON,0x0L);
                        }

                        /* Use new positions as old points
                           and start over */

                        pxy[0] = pxy[2];
                        pxy[1] = pxy[3];
                }
}               /* ends while */

set_clip(FALSE, &work_area);

graf_mouse(M_OFF, 0x0L);

                        /* Copy new screen to draw buffer,
                           replace mode (vro_cpyfm) */

        rast_op(3, &work_area, &scrn_mfdb, &draw_area, &draw_mfdb);
        graf_mouse(M_ON, 0x0L);
}


/*-------------------------------*/
/*      dr_code                  */
/*-------------------------------*/
WORD dr_code(pparms)
LONG    pparms;
{
                                /* Code to draw a box outline around the
                                   selected user defined object.  See the
                                   FARDRAW code.  */
        PARMBLK         pb;
        WORD            pxy[10], hl, wb;
        LONG            taddr;

        LBCOPY(ADDR(&pb), pparms, sizeof(PARMBLK));

        taddr = pb.pb_parm;
        userbrush_mfdb.mp = LLGET(BI_PDATA(taddr));
        hl = LWGET(BI_HL(taddr));
        wb = LWGET(BI_WB(taddr));
        userbrush_mfdb.fwp = wb << 3;
        userbrush_mfdb.fww = wb >> 1;
        userbrush_mfdb.fh = hl;
        userbrush_mfdb.np = 1;
        userbrush_mfdb.ff = 0;

        pxy[0] = pxy[1] = 0;
        pxy[2] = (wb << 3) - 1;
        pxy[3] = hl - 1;
        pxy[4] = pb.pb_x;
        pxy[5] = pb.pb_y;
        pxy[6] = pxy[4] + pb.pb_w - 1;
        pxy[7] = pxy[5] + pb.pb_h - 1;
```

```
		vrt_cpyfm(vdi_handle, 2, pxy, &userbrush_mfdb, &scrn_mfdb, usercolor);

		if((pb.pb_currstate!=pb.pb_prevstate)||(pb.pb_currstate&SELECTED))
		{
			if (pb.pb_currstate & SELECTED)
				vsl_color(vdi_handle,1);
			else
				vsl_color(vdi_handle,0);
			vsl_width(vdi_handle, 1);
			vsl_type(vdi_handle, FIS_SOLID);

					/* Draw a rectangle */
			pxy[0] = --pb.pb_x;
			pxy[1] = --pb.pb_y;
			pxy[2] = pb.pb_x + ++pb.pb_w - 1;
			pxy[3] = pb.pb_y + ++pb.pb_h - 1;
			pxy[4] = pxy[2];
			pxy[5] = pxy[3];
			pxy[3] = pxy[1];
			pxy[6] = pxy[0];
			pxy[7] = pxy[5];
			pxy[8] = pxy[0];
			pxy[9] = pxy[1];
			v_pline(vdi_handle, 5, pxy);
		}
		return (0);
}


/*------------------------------*/
/*	enab_menu		*/
/*------------------------------*/
VOID enab_menu(which)		/* Enable specified menu item	*/
WORD	which;
{
	undo_obj(addr_menu, which, DISABLED);
}


/*------------------------------*/
/*	eraser			*/
/*------------------------------*/
VOID
eraser(x, y)			/* Erase rectangle of eraser size at x,y */
WORD	x, y;
{
	WORD	erase_xy[4];

	if (demo_pen == PEN_FINE)
	{
		erase_xy[0] = x - 2;
		erase_xy[1] = y - 1;
		erase_xy[2] = x + 2;
		erase_xy[3] = y + 1;
	}
	else
		if (demo_pen == PEN_MEDIUM)
		{
			erase_xy[0] = x - 4;
			erase_xy[1] = y - 2;
```

```
				erase_xy[2] = x + 4;
				erase_xy[3] = y + 2;
			}
			else
			{
				erase_xy[0] = x - 6;
				erase_xy[1] = y - 3;
				erase_xy[2] = x + 6;
				erase_xy[3] = y + 3;
			}
	vr_recfl(vdi_handle, erase_xy);
}


/*-------------------------------*/
/*	get_file		*/
/*-------------------------------*/
WORD	get_file(loop)		/* Use file selector to get input file  */
BOOLEAN loop;
{
	WORD	fs_iexbutton;
	BYTE	fs_iinsel[13];

	while (TRUE)
	{
		get_path(file_name, "*.DOO");
		fs_iinsel[0] = '\0';

		fsel_input(ADDR(file_name), ADDR(fs_iinsel), &fs_iexbutton);
		if (fs_iexbutton)
		{
			add_file_name(file_name, fs_iinsel);
			file_handle = dos_open(ADDR(file_name),2);
			if (!loop || (loop && !DOS_ERR))
				return(1);
		}
		else
			return (0);
	}
}				/* end get_file */


/*-------------------------------*/
/*	get_parent		*/
/*-------------------------------*/
/*
	Routine that will find the parent of a given object.  The
	idea is to walk to the end of our siblings and return
	our parent.  If object is the root then return NIL as parent.
*/
WORD	get_parent(tree, obj)
LONG	tree;
WORD	obj;
{
	WORD	pobj;

	if (obj == NIL)  return (NIL);
	pobj = LWGET(OB_NEXT(obj));
	if (pobj != NIL)
	{
		while( LWGET(OB_TAIL(pobj)) != obj )
```

```
                {
                        obj = pobj;
                        pobj = LWGET(OB_NEXT(obj));
                }
        }
        return(pobj);
}


/*------------------------------*/
/*      get_path                */
/*------------------------------*/
VOID    get_path(tmp_path, spec)        /* Get directory path name      */
BYTE    *tmp_path, *spec;
{
        WORD    cur_drv;

        cur_drv = dos_gdrv();
        tmp_path[0] = cur_drv + 'A';
        tmp_path[1] = ':';
        tmp_path[2] = '\\';
        dos_gdir(cur_drv+1, ADDR(&tmp_path[3]));
        if (strlen(tmp_path) > 3)  strcat(tmp_path, "\\");
        else  tmp_path[2] = '\0';
        strcat(tmp_path, spec);
}


/*------------------------------*/
/*      get_select              */
/*------------------------------*/
WORD    get_select(tree, obj)
LONG    tree;
WORD    obj;
{
        WORD    nobj, cobj;
        LONG    bind, arry, temp;

        bind = LLGET(OB_SPEC(obj));
        dir_obj(tree, obj);
        LLSET(OB_SPEC(obj), LLGET(bind));
        arry = LLGET(bind + sizeof(LONG) );

        for (cobj = LWGET(OB_HEAD(obj)); cobj != obj;
        cobj = LWGET(OB_NEXT(cobj)))
        {
                dir_obj(tree, cobj);
                LLSET(OB_SPEC(cobj), LLGET(LLGET(OB_SPEC(cobj))));
        }

        nobj = LWGET(OB_NEXT(obj));
        dir_obj(tree, nobj);
        temp = LLGET(OB_SPEC(nobj));
        LLSET(OB_SPEC(nobj), LLGET(temp));
        return (WORD) (temp - arry) / sizeof(LONG) - 1;
}


/*------------------------------*/
/*      grect_to_array          */
/*------------------------------*/
```

```
VOID
grect_to_array(area, array)      /* convert x,y,w,h to upper left x,y &  */
GRECT   *area;                   /*                  lower right x,y     */
WORD    *array;
{
        *array++ = area->g_x;
        *array++ = area->g_y;
        *array++ = area->g_x + area->g_w - 1;
        *array = area->g_y + area->g_h - 1;
}


/*------------------------------*/
/*      hndl_button             */
/*------------------------------*/
VOID hndl_button()
{
                                /* A button event happened.
                                   If mouse position is inside the work
                                   area, then wait for mouse to move and then
                                   draw a line from old position to new. */

        if ( (mousex >= work_area.g_x) && (mousey >= work_area.g_y) &&
             (mousex < (work_area.g_x + work_area.g_w)) &&
             (mousey < (work_area.g_y + work_area.g_h)))

                        draw_pencil(mousex, mousey);
}



/*------------------------------*/
/*      hndl_dial               */
/*------------------------------*/
WORD    hndl_dial(tree, def, x, y, w, h)
LONG    tree;
WORD    def;
WORD    x, y, w, h;
{
        WORD    xdial, ydial, wdial, hdial, exitobj;
        WORD    xtype;

                                /* Get the coordinates of where the dialog
                                   would be when centered on the screen */

        form_center(tree, &xdial, &ydial, &wdial, &hdial);

                                /* Reserve the center of the screen */

        form_dial(0, x, y, w, h, xdial, ydial, wdial, hdial);

                                /* Draw and expanding rubber box from
                                   size of dialogue to center */

        form_dial(1, x, y, w, h, xdial, ydial, wdial, hdial);

                                /* Display the dialog */

        objc_draw(tree, ROOT, MAX_DEPTH, xdial, ydial, wdial, hdial);

        FOREVER                 /* Get the user's input */
        {
                exitobj = form_do(tree, def) & 0x7FFF;
                                /* Mask out any non-user defined objects.
                                   All user defined objects have high byte. */

                xtype = LWGET(OB_TYPE(exitobj)) & 0xFF00;

                                /* Get out if not user defined */

                if (!xtype) break;

                                /* Go to user defined interpreter */

                xtend_do(tree, exitobj, xtype);
        }
                                /* Draws shrinking box */

        form_dial(2, x, y, w, h, xdial, ydial, wdial, hdial);

                                /* Release the reserved screen space */

        form_dial(3, x, y, w, h, xdial, ydial, wdial, hdial);

        return (exitobj);
}


/*------------------------------*/
/*      hndl_menu               */
/*------------------------------*/
WORD
hndl_menu(title, item)
WORD    title, item;
{
        LONG    tree;
        GRECT   box;

                                /* A menu item was selected.  */

        graf_mouse(ARROW, 0x0L);
        switch (title) {
         case DEMODESK:

                if (item == DEMOINFO) {
                                /* display Demo Info...
                                   Find the coordinates of the Desk menu
                                   title relative to the screen.  Only used
                                   to show an expanding/shrinking box. */

                        objc_xywh(addr_menu, DEMODESK, &box);

                                /* Get the address of the info dialog */

                        rsrc_gaddr(R_TREE, DEMOINFD, &tree);

                                /* Display the dialog */

                        hndl_dial(tree, 0, box.g_x, box.g_y, box.g_w, box.g_h)

                                /* Re-initialize the object */
                        desel_obj(tree, DEMOOK);
                }
                break;

         case DEMOFILE:
                switch (item) {
                 case DEMOLOAD:
                        do_load(TRUE);
                        break;
                 case DEMOSAVE:
                        do_save();
                        break;
                 case DEMOSVAS:
                        do_svas();
                        break;
                 case DEMOABAN:
                        file_handle = dos_open(ADDR(file_name),2);
                        do_load(FALSE);
                        break;
                 case DEMOQUIT:
                        return(TRUE);
                }

         case DEMOOPTS:
                switch (item) {
                 case DEMOPENS:
                        do_penselect();
                        break;
                 case DEMOERAP:

                                /* Clear the screen and the draw buffer.
                                   As we are always erasing the entire window
                                   use scrn_area & scrn_mfdb for the source. *

                        rast_op(0, &scrn_area, &scrn_mfdb,
                                &scrn_area, &draw_mfdb);
                        restore_work();

                        break;
                }
        }
        menu_tnormal(addr_menu,title,TRUE);
        graf_mouse(monumber, mofaddr);
        return (FALSE);
}


/*------------------------------*/
/*      hndl_mouse              */
/*------------------------------*/
VOID hndl_mouse()
{
        if (m_out)  graf_mouse(ARROW, 0x0L);
        else        graf_mouse(monumber, mofaddr);

        m_out = !m_out;
}
```

```
/*------------------------------*/
/*      hndl_msg                */
/*------------------------------*/
BOOLEAN hndl_msg()
{
        WORD    wdw_hndl;
        GRECT   work;

        wdw_hndl = msg_buff[3]; /* Get window handle of any message that
                                   may be window related. */

        switch( msg_buff[0] )   /* Message type  */
        {
          case MN_SELECTED:     /* Mouse moved to menu */

                return(hndl_menu(wdw_hndl, msg_buff[4]));

          case WM_REDRAW:       /* Window needs to be re-drawn
                                   msg_buff[4-7] = GRECT (x,y,w,h) */

                do_redraw(wdw_hndl, (GRECT *) &msg_buff[4]);
                break;

          case WM_TOPPED:       /* Place window on top */

                wind_set(wdw_hndl, WF_TOP, 0, 0, 0, 0);
                break;

          case WM_CLOSED:       /* Close the window */

                return(TRUE);

          case WM_FULLED:       /* Full button clicked */

                do_full(wdw_hndl);
                break;

          case WM_ARROWED:      /* Mouse touched slider area */

                switch(msg_buff[4])
                {
                    case WA_UPPAGE:
                        draw_area.g_y = max(draw_area.g_y - draw_area.g_h, 0);
                        break;

                    case WA_DNPAGE:
                        draw_area.g_y += draw_area.g_h;
                        break;

                    case WA_UPLINE:
                        draw_area.g_y = max(draw_area.g_y - YSCALE(16), 0);
                        break;

                    case WA_DNLINE:
                        draw_area.g_y += YSCALE(16);
                        break;

                    case WA_LFPAGE:     /* Page left */
                        draw_area.g_x = max(draw_area.g_x-draw_area.g_w, 0);
                        break;

                    case WA_RTPAGE:     /* Page right */
                        draw_area.g_x += draw_area.g_w;
                        break;
          case WA_LFLINE:        /* Column left */
                draw_area.g_x = max(draw_area.g_x - 16, 0);
                break;

          case WA_RTLINE:        /* Column Right */
                draw_area.g_x += 16;
                break;
        }
                          /* Get the current work area */

        set_work(TRUE);
                          /* Restore the work area from draw buffer */

        restore_work();
        break;

    case WM_HSLID:        /* Horizontal slider position changed */

        draw_area.g_x = align_x(UMUL_DIV(draw_mfdb.fwp - draw_area.g_w
                                     msg_buff[4], 1000));
        set_work(TRUE); /* Get new work area and update slider */
        restore_work();
        break;

    case WM_VSLID:        /* Vertical slider position changed */

        draw_area.g_y = UMUL_DIV(draw_mfdb.fh - draw_area.g_h,
                               msg_buff[4],1000);
        set_work(TRUE);
        restore_work();
        break;

    case WM_SIZED:        /* Size button clicked.
                             The new window size must be calculated.
                             Use wind_calc instead of wind_get because
                             wind_calc obtains the possible sizes,
                             whereas wind_get obtains the actual.
                             Besides the values obtained by wind_get
                             are not valid until we change them as the
                             window size has just been changed.

                             Calculate the size of the new work area */

        wind_calc(1, 0x0fef, msg_buff[4], msg_buff[5], msg_buff[6],
             msg_buff[7], &work.g_x, &work.g_y, &work.g_w,
            &work.g_h);
        work.g_x = align_x(work.g_x);
        work.g_w = align_x(work.g_w);
                          /* Set the size of the current window, which
                             automatically updates the previous and
                             work figures.  */

        wind_set(wdw_hndl, WF_CXYWH, msg_buff[4],
              msg_buff[5], msg_buff[6], msg_buff[7]);

        set_work(TRUE);
        break;
        case WM_MOVED:          /* Window has been moved */

                msg_buff[4] = align_x(msg_buff[4]);
                wind_set(wdw_hndl, WF_CXYWH, align_x(msg_buff[4]) - 1,
                         msg_buff[5], msg_buff[6], msg_buff[7]);
                set_work(FALSE);
                break;

        }                       /* End switch */
        return(FALSE);          /* If hndl_msg returns TRUE then it means
                                   user clicked on close box or wants to
                                   quit.  */
}                               /* End hndl_msg */


/*------------------------------*/
/*      indir_obj               */
/*------------------------------*/
VOID    indir_obj(tree, which)
LONG    tree;
WORD    which;
{
        WORD    flags;

        flags = LWGET(OB_FLAGS(which));
        LWSET(OB_FLAGS(which),  flags | INDIRECT);
}

/*------------------------------*/
/*      max                     */
/*------------------------------*/
WORD    max(a, b)               /* Return max of two values */
WORD        a, b;
{
        return( (a > b) ? a : b );
}


/*------------------------------*/
/*      min                     */
/*------------------------------*/
WORD    min(a, b)               /* Return min of two values */
WORD        a, b;
{
        return( (a < b) ? a : b );
}


/*------------------------------*/
/*      move_do                 */
/*------------------------------*/
VOID    move_do(tree, obj, inc)
LONG    tree;
WORD    obj, inc;
{
                                /* This routine shifts the color selection
                                   panel depending on whether the forward
                                   or the backward arrow was selected.
                                   See the commentary for set_selection */
        WORD    cobj;
        LONG    n, bind, arry, limit, obspec;

        obj = get_parent(tree, obj);
        obj = LWGET(OB_NEXT(obj));
        bind = LLGET(OB_SPEC(obj));
        arry = LLGET(bind + sizeof(LONG));
        n = LLGET(arry) * sizeof(LONG);
        limit = arry + n;

        for (cobj = LWGET(OB_HEAD(obj)); cobj != obj;
        cobj = LWGET(OB_NEXT(cobj)))
        {
                obspec = LLGET(OB_SPEC(cobj));
                obspec += inc * sizeof(LONG);
                while (obspec <= arry || obspec > limit)
                        obspec += n * ((obspec > limit)? -1: 1);
                LLSET(OB_SPEC(cobj), obspec);
        }

        redraw_do(tree, obj);
}


/*------------------------------*/
/*      pict_init               */
/*------------------------------*/

VOID pict_init() {

        LONG    tree;
        WORD    tr_obj, nobj;

                                /* Initialize all the user defined objects.

                                   Get the addr of the Info dialog  and
                                   transform its G_IMAGE into device specfic
                                   format.  */

        rsrc_gaddr(R_TREE, DEMOINFD, &tree);
        trans_gimage(tree, DEMOIMG);

                                /* There are 6 additional G_IMAGEs in DEMO.
                                   They are used in the PENCIL/ERASER dialog
                                   in the OPTIONS menu.  When this dialog
                                   is displayed you will notice that there
                                   3 pencil and 3 eraser thicknesses
                                   (6 in all).  DEMO wants to highlight these
                                   by drawing a small box around the selected
                                   one.
                                   First change them to user defined objects,
                                   transform them to device specific format,
                                   and initialize the APPLBLKs.  See dr_code
                                   and FARDRAW.  */

        rsrc_gaddr(R_TREE, DEMOPEND, &tree);
        for (tr_obj = DEMOPFIN; tr_obj <= DEMOEBRD; tr_obj++){

                                /* Transform the image to device specific */

                trans_gimage(tree, tr_obj);
```

```
                                /* Set the object type field to user defined *

                LWSET(OB_TYPE(tr_obj), G_USERDEF);

                nobj = tr_obj - DEMOPFIN;

                                /* Save the address of the external routine */

                brushub[nobj].ub_code = drawaddr;

                                /* Use the object spec which points to the
                                   BITBLK that describes the G_IMAGE as the
                                   parameter to drawaddr. */

                brushub[nobj].ub_parm = LLGET(OB_SPEC(tr_obj));

                                /* Now set the object spec to point to the
                                   address of the appropriate APPLBLK */

                LLSET(OB_SPEC(tr_obj), ADDR(&brushub[nobj]));
        }
}


/*------------------------------*/
/*      objc_xywh               */
/*------------------------------*/
VOID    objc_xywh(tree, obj, p)         /* get x,y,w,h for specified object */
LONG    tree;
WORD    obj;
GRECT   *p;
{
        objc_offset(tree, obj, &p->g_x, &p->g_y);
        p->g_w = LWGET(OB_WIDTH(obj));
        p->g_h = LWGET(OB_HEIGHT(obj));
}


/*------------------------------*/
/*      rast_op                 */
/*------------------------------*/
VOID
rast_op(mode, s_area, s_mfdb, d_area, d_mfdb)   /* bit block level trns */
WORD    mode;
GRECT   *s_area, *d_area;
MFDB    *s_mfdb, *d_mfdb;
{
        WORD    pxy[8];

        grect_to_array(s_area, pxy);
        grect_to_array(d_area, &pxy[4]);
        vro_cpyfm(vdi_handle, mode, pxy, s_mfdb, d_mfdb);
}


/*------------------------------*/
/*      rc_copy                 */
/*------------------------------*/
VOID
rc_copy(psbox, pdbox)           /* copy source to destination rectangle */
GRECT   *psbox;
GRECT   *pdbox;
```

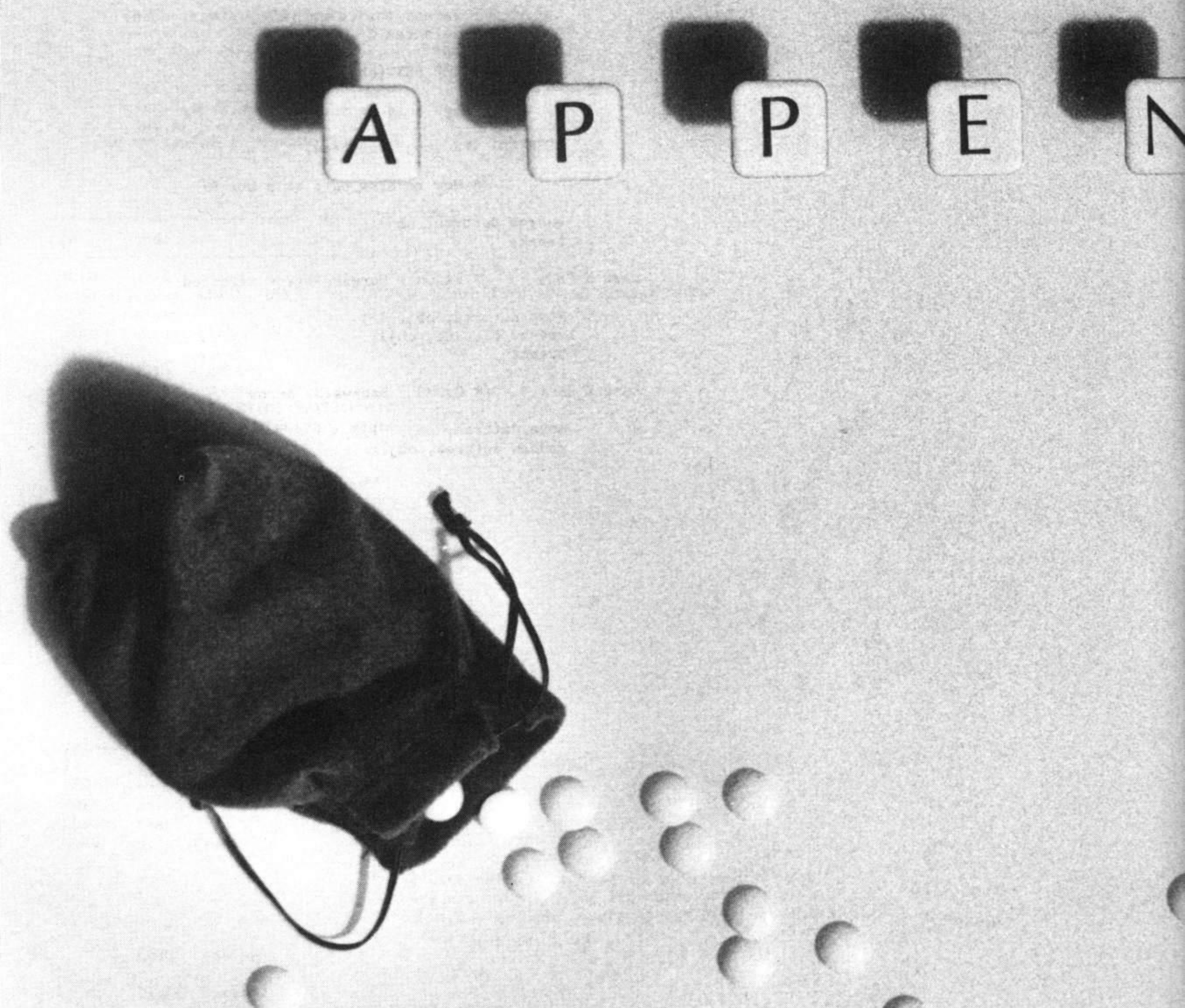
```
{
        pdbox->g_x = psbox->g_x;
        pdbox->g_y = psbox->g_y;
        pdbox->g_w = psbox->g_w;
        pdbox->g_h = psbox->g_h;
}


/*------------------------------*/
/*      rc_equal                */
/*------------------------------*/
WORD
rc_equal(p1, p2)                /* tests for two equal rectangles */
GRECT           *p1, *p2;
{
        if ((p1->g_x != p2->g_x) ||
            (p1->g_y != p2->g_y) ||
            (p1->g_w != p2->g_w) ||
            (p1->g_h != p2->g_h))
                return(FALSE);
        return(TRUE);
}


/*------------------------------*/
/*      rc_intersect            */
/*------------------------------*/
WORD
rc_intersect(p1, p2)            /* compute intersect of two rectangles  */
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) );
}


/*------------------------------*/
/*      redraw_do               */
/*------------------------------*/
VOID    redraw_do(tree, obj)
LONG    tree;
WORD    obj;
{
        GRECT   o;

        objc_xywh(tree, obj, &o);
        o.g_x -= 3; o.g_y -= 3; o.g_w += 6; o.g_h += 6;
        objc_draw(tree, ROOT, MAX_DEPTH, o.g_x, o.g_y, o.g_w, o.g_h);
}
```

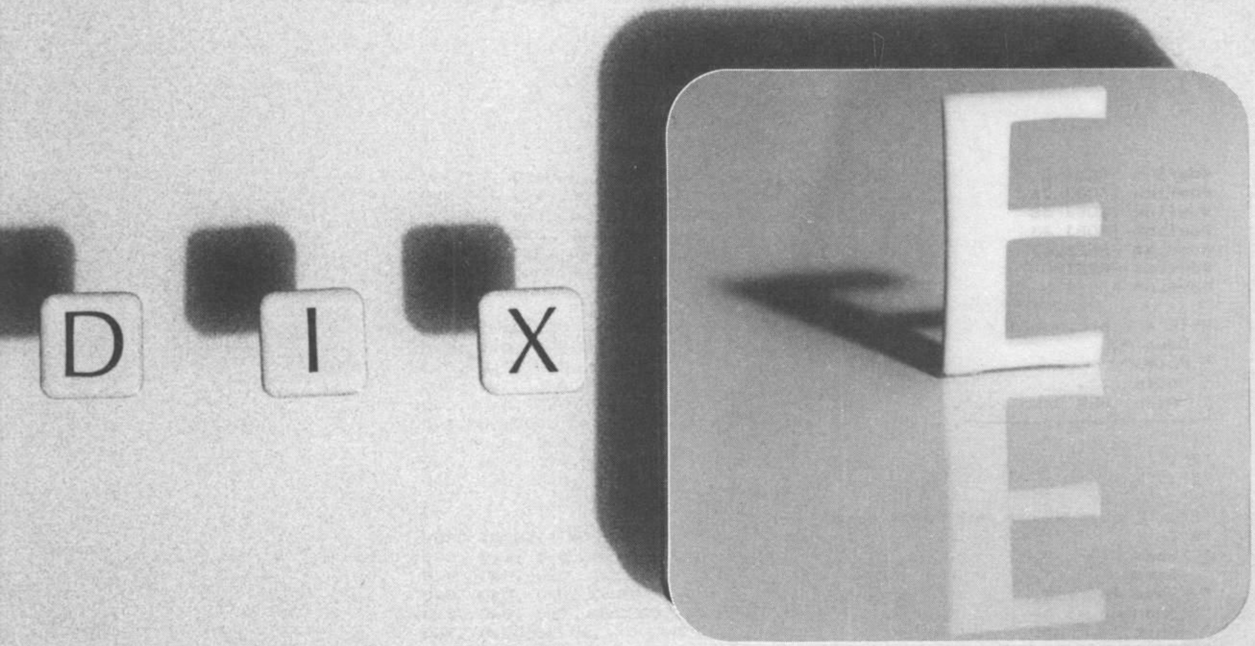
```
/*------------------------------*/
/*      restore_work            */
/*------------------------------*/
VOID    restore_work()          /* restore work_area from draw_area     */
{
        graf_mouse(M_OFF, 0x0L);
        rast_op(3, &draw_area, &draw_mfdb, &work_area, &scrn_mfdb);
        graf_mouse(M_ON, 0x0L);
}


/*------------------------------*/
/*      sel_obj                 */
/*------------------------------*/
VOID    sel_obj(tree, which)    /* turn on selected bit of specfied object */
LONG    tree;
WORD    which;
{
        WORD    state;

        state = LWGET(OB_STATE(which));
        LWSET(OB_STATE(which), state | SELECTED);
}


/*------------------------------*/
/*      set_clip                */
/*------------------------------*/
VOID    set_clip(clip_flag, s_area)     /* set clip to specified area   */
WORD    clip_flag;
GRECT   *s_area;
{
        WORD    pxy[4];

        grect_to_array(s_area, pxy);
        vs_clip(vdi_handle, clip_flag, pxy);
}


/*------------------------------*/
/*      set_eraser              */
/*------------------------------*/
VOID    set_eraser(pen, height, eraser)
WORD    pen, height;
BYTE    *eraser;
{
        demo_pen = pen;
        demo_height = height;
        demo_shade = PEN_ERASER;
        monumber = 255;
        mofaddr = ADDR(eraser);
}


/*------------------------------*/
/*      set_pen                 */
/*------------------------------*/
VOID    set_pen(pen, height)
WORD    pen, height;
{
        demo_pen = pen;
        demo_height = height;
        monumber = 5;
        mofaddr = 0x0L;
}


/*------------------------------*/
/*      set_select              */
/*------------------------------*/
VOID    set_select(tree, obj, init_no, bind, arry)
LONG    tree, bind[], arry[];
WORD    obj, init_no;
{
                                /* Sets the address of the color selection
                                   array into the parent of the color
                                   selection panel object.  */

        WORD    n, nobj, cobj, count;

                                /* First make the flag field INDIRECT */
        indir_obj(tree, obj);

                                /* Save the old object spec into a local */

        bind[0] = LLGET(OB_SPEC(obj));

                                /* Point to the local */

        LLSET(OB_SPEC(obj), ADDR(bind));

                                /* Point to the color array */
        bind[1] = ADDR(arry);

                                /* Set the first 4 TOUCHEXIT objects to the
                                   appropriate color.  */
        n = (WORD) arry[0];
        count = 0;
        for (cobj = LWGET(OB_HEAD(obj)); cobj != obj;
        cobj = LWGET(OB_NEXT(cobj)))
        {
                indir_obj(tree, cobj);
                LLSET(OB_SPEC(cobj), ADDR( &arry[count + 1] ));
                count = (count + 1) % n;
        }

                                /* Set the selected G_BOXCHAR to the
                                   appropriate color and make it INDIRECT */

        nobj = LWGET(OB_NEXT(obj));
        indir_obj(tree, nobj);
        LLSET(OB_SPEC(nobj), ADDR( &arry[1 + init_no % n] ));
}


/*------------------------------*/
/*      set_work                */
/*------------------------------*/
VOID    set_work(slider_update) /* update draw area, clamping to page    */
BOOLEAN slider_update;          /*   edges, and update sliders if req'd */
{
        WORD    i;
```

```
                                        /* Find size of work area */

                wind_get(demo_whndl, WF_WXYWH,&work_area.g_x, &work_area.g_y,
                        &work_area.g_w, &work_area.g_h);

                                        /* Clamp work area to page edges */

                draw_area.g_x = align_x(draw_area.g_x);
                if ((i = draw_mfdb.fwp - (draw_area.g_x + work_area.g_w)) < 0)
                        draw_area.g_x += i;
                if ((i = draw_mfdb.fh - (draw_area.g_y + work_area.g_h)) < 0)
                        draw_area.g_y += i;

                if (slider_update)
                {
                wind_set(demo_whndl, WF_HSLIDE, UMUL_DIV(draw_area.g_x, 1000,
                        draw_mfdb.fwp - work_area.g_w), 0, 0, 0);
                wind_set(demo_whndl, WF_VSLIDE, UMUL_DIV(draw_area.g_y, 1000,
                        draw_mfdb.fh - work_area.g_h), 0, 0, 0);
                wind_set(demo_whndl, WF_HSLSIZ, UMUL_DIV(work_area.g_w, 1000,
                        draw_mfdb.fwp), 0, 0, 0);
                wind_set(demo_whndl, WF_VSLSIZ, UMUL_DIV(work_area.g_h, 1000,
                        draw_mfdb.fh), 0, 0, 0);
        }

                                /* Only use portion of work_area on screen */

        rc_intersect(&scrn_area, &work_area);
        draw_area.g_w = work_area.g_w;
        draw_area.g_h = work_area.g_h;
}


/*------------------------------*/
/*      string_addr             */
/*------------------------------*/
LONG    string_addr(which)      /* Returns a tedinfo LONG string addr   */
WORD    which;
{
        LONG    where;

        rsrc_gaddr(R_STRING, which, &where);
        return (where);
}


/*------------------------------*/
/*      trans_gimage            */
/*      transform a standard format to a device specific format */
/*------------------------------*/
VOID trans_gimage(tree, obj)
        LONG    tree;
        WORD    obj;
{
        LONG    obspec;
        WORD    wb;
        MFDB    src, dst;

        obspec = LLGET(OB_SPEC(obj));
        src.mp = dst.mp = LLGET(BI_PDATA(obspec));
```

```
	wb = LWGET(BI_WB(obspec));
	src.fwp = dst.fwp = wb<<3;
	src.fww = dst.fww = wb>>1;
	src.fh = dst.fh = LWGET(BI_HL(obspec));

	src.np = dst.np = 1;
	src.ff = TRUE;
	dst.ff = FALSE;
	vr_trnfm(vdi_handle, &src, &dst );
}

/*------------------------------*/
/*	undo_obj		*/
/*------------------------------*/
VOID
undo_obj(tree, which, bit)	/* clear specified bit in object state	*/
LONG	tree;
WORD	which, bit;
{
	WORD	state;

	state = LWGET(OB_STATE(which));
	LWSET(OB_STATE(which), state & ~bit);
}

/*------------------------------*/
/*	unflag_obj		*/
/*------------------------------*/
VOID	unflag_obj(tree, which, bit)
LONG	tree;
WORD	which, bit;
{
	WORD	flags;

	flags = LWGET(OB_FLAGS(which));
	LWSET(OB_FLAGS(which), flags & ~bit);
}


/*------------------------------*/
/*	xtend_do		*/
/*------------------------------*/
VOID	xtend_do(tree, obj, xtype)
LONG	tree;
WORD	obj, xtype;
{
				/* The user defined command interpreter
				There are only 3 types (forward,
				backward arrow and an actual color selected
				*/
	LONG	obspec;

	switch (xtype) {
		case X_SEL:	/* 0x300 - Color selected  */

				/* Get the color pointed at */

			obspec = LLGET(OB_SPEC(obj));

				/* Go back up the tree */
```

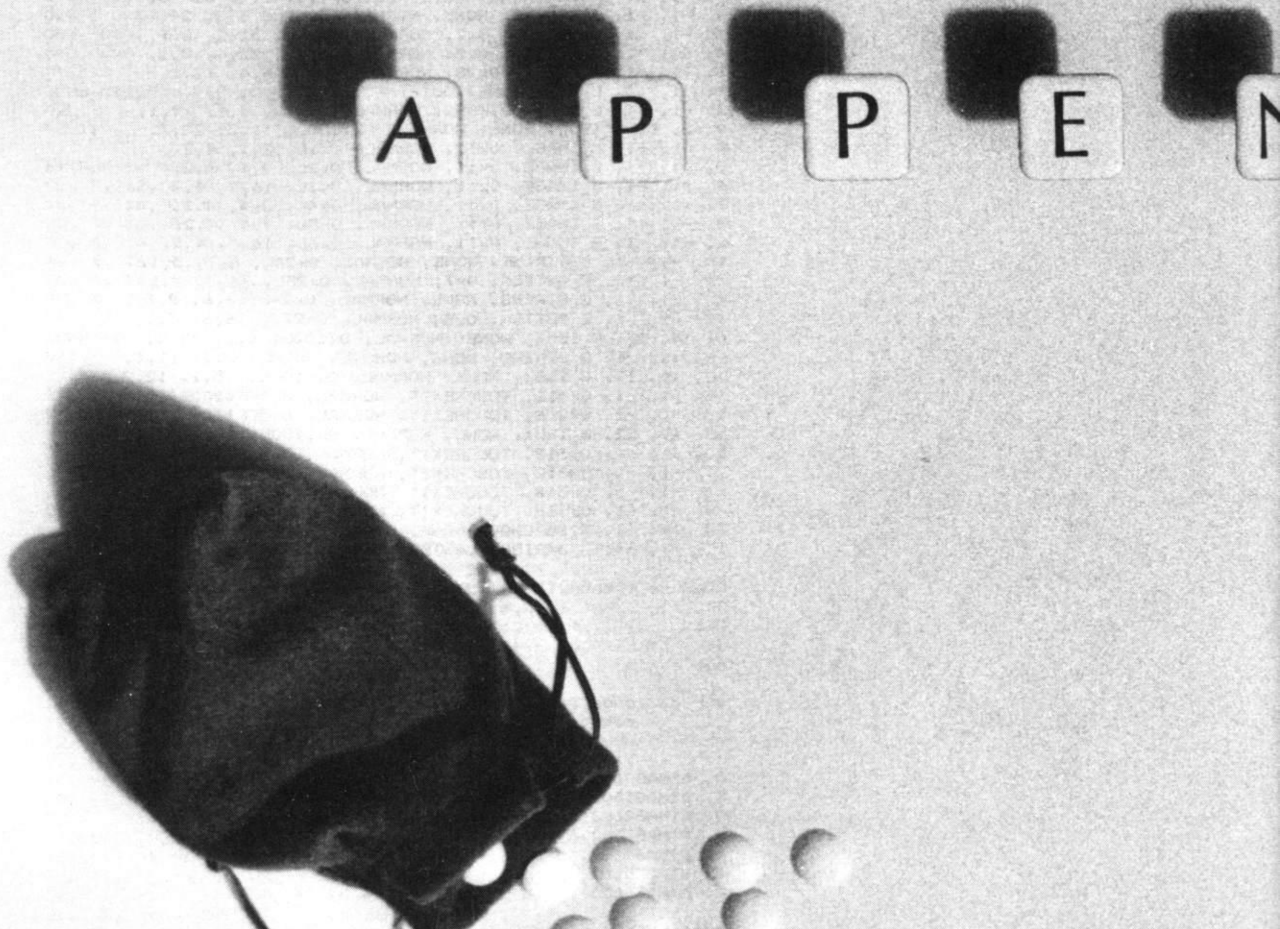
```
                    obj = get_parent(tree, obj);
                            /* Parent points to next brother, color
                               selected G_BOXCHAR */

                    obj = LWGET(OB_NEXT(obj));

                            /* Set this to the saved color */

                    LLSET(OB_SPEC(obj), obspec);

                            /* Now re-draw only this box */

                    redraw_do(tree, obj);
                    break;

            case X_FWD:     /* 0x100 - Forward arrow selected */

                    move_do(tree, obj, 1);
                    redraw_do(tree, obj);
                    break;

            case X_BAK:     /* 0x200 - Backwards arrow  */

                    move_do(tree, obj, -1);
                    redraw_do(tree, obj);
        }
}
```


A P P E N

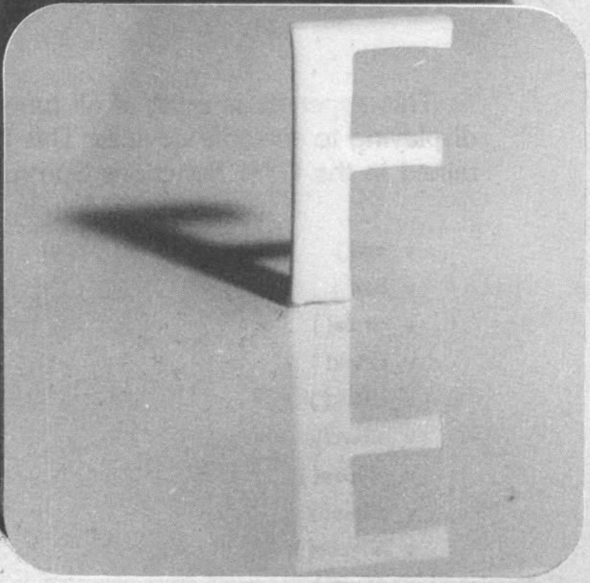


LISTING OF .RSC OUTPUT FILE FOR DEMO

```
#define T0OBJ 0
#define T1OBJ 26
#define T2OBJ 38
#define T3OBJ 43
#define FREEBB 7
#define FREEIMG 7
#define FREESTR 42

BYTE *rs_strings[] = {
" Desk ",
" File ",
" Options ",
"  About GEM Demo ...",
"--------------------",
"1",
"2",
"3",
"4",
"5",
"6",
"  Load       ",
"  Save       ",
"  Save As... ",
"  Abandon    ",
"  Quit       ",
"  Pen \ Eraser Selection ",
"-------------------------",
"  Erase Picture          ",
"GEM Demo ",
"GEM Sample Application",
"Authors",
"-------",
"Ok",
"Tom Rolander",
"Tim Oren",
"Phillip Balma",
"Version 1.2, February, 1986",
"Digital Research, Inc.",
"Save GEM Demo picture as",
"Ok",
"Cancel",
"",
"________.DOO",
"ANNNNNNN",
"GEM Demo Pen/Eraser Selection",
"Pens:",
"Ok",
"Erasers:",
"Cancel",
"Pen Colors:",
"Selected:",
"[2][You are about|to write over|an existing file.][Ok| Cancel ]",
"[3][ GEM does not have| any windows left,| Dr Doodle aborting][ Sorry\
 ]"};
                                    /* Image Block for DRI logo /
WORD IMAG0[] = {
0x0, 0x0, 0x3FFF, 0xFFF8,
0x7FFF, 0xFFFC, 0xE000, 0xE,
0xC000, 0x6, 0xC000, 0x6,
0xC3F8, 0x3F86, 0xC7FC, 0x7FC6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
0xC7FC, 0x60C6, 0xC7FC, 0x60C6,
'OxC7FC, Ox60C6, OxC7FC, Ox60C6,
 OxC7FC, Ox60C6, OxC7FC, Ox60C6,
 OxC7FC, Ox7FC6, OxC3F8, Ox3F86,
 OxC000, Ox6, OxC000, Ox6,
 OxE000, OxE, Ox7FFF, OxFFFC,
 Ox3FFF, OxFFF8, Ox0, Ox0};

                                        /* Image block for fine pencil */
WORD IMAG1[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox3, Ox8000, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0};

                                        /* Image Block for medium pencil */
WORD IMAG2[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox7, OxC000,
OxF, OxE000, Ox7, OxC000,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0};

                                        /* Image Block for broad pencil */
WORD IMAG3[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox7, OxC000, Ox1F, OxF000,
Ox3F, OxF800, Ox1F, OxF000,
Ox7, OxC000, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0};

                                        /*Image Block for fine eraser */
WORD IMAG4[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox7, OxC000, Ox4, Ox4000,
Ox7, OxC000, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0};

                                        /* Image Block for medium eraser */
WORD IMAG5[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox1F, OxF000,
Ox10, Ox1000, Ox10, Ox1000,
Ox10, Ox1000, Ox1F, OxF000,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0};

                                        /* Image Block for broad eraser */
WORD IMAG6[] = {
Ox0, Ox0, Ox0, Ox0,
Ox0, Ox0, Ox0, Ox0,
0x7F, 0xFC00, 0x40, 0x400,
0x40, 0x400, 0x40, 0x400,
0x40, 0x400, 0x40, 0x400,
0x7F, 0xFC00, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0};

LONG rs_frstr[] = {
42L,
43L};

BITBLK rs_bitblk[] = {
0L, 4, 32, 0, 0, 1,
1L, 4, 16, 0, 0, 1,
2L, 4, 16, 0, 0, 1,
3L, 4, 16, 0, 0, 1,
4L, 4, 16, 0, 0, 1,
5L, 4, 16, 0, 0, 1,
6L, 4, 16, 0, 0, 1};

LONG rs_frimg[] = {
0};

ICONBLK rs_iconblk[] = {
0};

TEDINFO rs_tedinfo[] = {
32L, 33L, 34L, 3, 6, 0, 0x1180, 0x0, -1, 1,13};

OBJECT rs_object[] = {
-1, 1, 6, G_IBOX, NONE, NORMAL, 0x0L, 0,0, 80,25,
6, 2, 2, G_BOX, NONE, NORMAL, 0x1100L, 0,0, 80,513,
1, 3, 5, G_IBOX, NONE, NORMAL, 0x0L, 2,0, 21,769,
4, -1, -1, G_TITLE, NONE, NORMAL, 0x0L, 0,0, 6,769,
5, -1, -1, G_TITLE, NONE, NORMAL, 0x1L, 6,0, 6,769,
2, -1, -1, G_TITLE, NONE, NORMAL, 0x2L, 12,0, 9,769,
0, 7, 22, G_IBOX, NONE, NORMAL, 0x0L, 0,769, 80,19,
16, 8, 15, G_BOX, NONE, NORMAL, 0xFF1100L, 2,0, 20,8,
9, -1, -1, G_STRING, SELECTABLE, NORMAL, 0x3L, 0,0, 20,1,
10, -1, -1, G_STRING, NONE, DISABLED, 0x4L, 0,1, 20,1,
11, -1, -1, G_STRING, NONE, NORMAL, 0x5L, 0,2, 20,1,
12, -1, -1, G_STRING, NONE, NORMAL, 0x6L, 0,3, 20,1,
13, -1, -1, G_STRING, NONE, NORMAL, 0x7L, 0,4, 20,1,
14, -1, -1, G_STRING, NONE, NORMAL, 0x8L, 0,5, 20,1,
15, -1, -1, G_STRING, NONE, NORMAL, 0x9L, 0,6, 20,1,
7, -1, -1, G_STRING, NONE, NORMAL, 0xAL, 0,7, 20,1,
22, 17, 21, G_BOX, NONE, NORMAL, 0xFF1100L, 8,0, 13,5,
18, -1, -1, G_STRING, SELECTABLE, NORMAL, 0xBL, 0,0, 13,1,
19, -1, -1, G_STRING, NONE, DISABLED, 0xCL, 0,1, 13,1,
20, -1, -1, G_STRING, SELECTABLE, NORMAL, 0xDL, 0,2, 13,1,
21, -1, -1, G_STRING, SELECTABLE, NORMAL, 0xEL, 0,3, 13,1,
16, -1, -1, G_STRING, SELECTABLE, NORMAL, 0xFL, 0,4, 13,1,
6, 23, 25, G_BOX, NONE, NORMAL, 0xFF1100L, 14,0, 25,3,
24, -1, -1, G_STRING, NONE, NORMAL, 0x10L, 0,0, 25,1,
25, -1, -1, G_STRING, NONE, DISABLED, 0x11L, 0,1, 25,1,
22, -1, -1, G_STRING, LASTOB, NORMAL, 0x12L, 0,2, 25,1,
-1, 1, 11, G_BOX, NONE, OUTLINED, 0x21100L, 8,9, 38,15,
2, -1, -1, G_STRING, NONE, NORMAL, 0x13L, 14,1, 9,1,
3, -1, -1, G_STRING, NONE, NORMAL, 0x14L, 7,2, 22,1,
4, -1, -1, G_IMAGE, NONE, NORMAL, 0x0L, 2,3, 4,4,
5, -1, -1, G_STRING, NONE, NORMAL, 0x15L, 14,4, 7,1,
6, -1, -1, G_STRING, NONE, NORMAL, 0x16L, 14,5, 7,1,
7, -1, -1, G_BUTTON, 0x7, NORMAL, 0x17L, 27,5, 8,1,
8, -1, -1, G_STRING, NONE, NORMAL, 0x18L, 11,6, 12,1,
9, -1, -1, G_STRING, NONE, NORMAL, 0x19L, 13,7, 8,1,
10, -1, -1, G_STRING, NONE, NORMAL, 0x1AL, 11,8, 13,1,
11, -1, -1, G_STRING, NONE, NORMAL, 0x1BL, 5,11, 27,1,
0, -1, -1, G_STRING, LASTOB, NORMAL, 0x1CL, 8,12, 22,1,
-1, 1, 4, G_BOX, NONE, OUTLINED, 0x21100L, 0,0, 43,7,
2, -1, -1, G_STRING, NONE, NORMAL, 0x1DL, 3,2, 24,1,
3, -1, -1, G_BUTTON, 0x7, NORMAL, 0x1EL, 32,2, 8,1,
4, -1, -1, G_BUTTON, 0x5, NORMAL, 0x1FL, 32,4, 8,1,
0, -1, -1, G_FTEXT, 0x28, NORMAL, 0x0L, 8,4, 12,1,
-1, 1, 13, G_BOX, NONE, OUTLINED, 0x21100L, 8,9, 47,13,
2, -1, -1, G_STRING, NONE, NORMAL, 0x23L, 8,1, 29,1,
9, 3, 8, G_IBOX, NONE, NORMAL, 0x1100L, 11,2, 22,7,
4, -1, -1, G_IMAGE, 0x11, NORMAL, 0x1L, 2,1, 4,2,
5, -1, -1, G_IMAGE, 0x11, NORMAL, 0x2L, 9,1, 4,2,
6, -1, -1, G_IMAGE, 0x11, NORMAL, 0x3L, 16,1, 4,2,
7, -1, -1, G_IMAGE, 0x11, NORMAL, 0x4L, 2,4, 4,2,
8, -1, -1, G_IMAGE, 0x11, NORMAL, 0x5L, 9,4, 4,2,
2, -1, -1, G_IMAGE, 0x11, NORMAL, 0x6L, 16,4, 4,2,
10, -1, -1, G_STRING, NONE, NORMAL, 0x24L, 6,3, 5,1,
11, -1, -1, G_BUTTON, 0x7, NORMAL, 0x25L, 36,3, 7,1,
12, -1, -1, G_STRING, NONE, NORMAL, 0x26L, 3,6, 8,1,
13, -1, -1, G_BUTTON, 0x5, NORMAL, 0x27L, 36,6, 7,1,
0, 14, 24, G_IBOX, NONE, NORMAL, 0x100L, 4,8, 29,5,
15, -1, -1, G_STRING, NONE, NORMAL, 0x28L, 2,1, 11,1,
18, 16, 17, G_IBOX, NONE, NORMAL, 0x1100L, 15,1, 12,2,
17, -1, -1, 0x11B, TOUCHEXIT, NORMAL, 0x4FF1100L, 0,0, 2,1,
15, -1, -1, 0x21B, TOUCHEXIT, NORMAL, 0x3FF1100L, 10,0, 2,1,
23, 19, 22, G_IBOX, NONE, NORMAL, 0x1100L, 17,1, 8,1,
20, -1, -1, 0x31B, TOUCHEXIT, NORMAL, 0x31FF1071L, 0,0, 2,1,
21, -1, -1, 0x31B, TOUCHEXIT, NORMAL, 0x32FF1072L, 2,0, 2,1,
22, -1, -1, 0x31B, TOUCHEXIT, NORMAL, 0x33FF1073L, 4,0, 2,1,
18, -1, -1, 0x31B, TOUCHEXIT, NORMAL, 0x34FF1074L, 6,0, 2,1,
24, -1, -1, G_BOXCHAR, NONE, NORMAL, 0x31FF1071L, 15,3, 2,1,
13, -1, -1, G_STRING, LASTOB, NORMAL, 0x29L, 4,3, 9,1};

LONG rs_trindex[] = {
0L,
26L,
38L,
43L};

struct foobar {
        WORD    dummy;
        WORD    *image;
        } rs_imdope[] = {
0, &IMAG0[0],
0, &IMAG1[0],
0, &IMAG2[0],
0, &IMAG3[0],
0, &IMAG4[0],
0, &IMAG5[0],
0, &IMAG6[0]};

#define NUM_STRINGS 44
#define NUM_FRSTR 2
#define NUM_IMAGES 7
#define NUM_BB 7
#define NUM_FRIMG 0
#define NUM_IB 0
#define NUM_TI 1
#define NUM_OBS 68
#define NUM_TREE 4

BYTE pname[] = "DEMO.RSC";
```


APPEN

D I X



FUNCTIONS AVAILABLE IN METAFILES

This appendix is a list of all functions you can use when displaying to metafile devices. This information is also contained in the GEM Functions Summary in Appendix B.

v_arc( )
v_bar( )
v_circle( )
v_clrwk( )
v_clswk( )
v_ellarc( )
v_ellipse( )
v_ellpie( )
v_fillarea( )
v_gtext( )
v_justified( )
v_opnwk( )
v_pieslice
v_pline( )
v_pmarker( )
v_rbox( )
v_rfbox( )
vm_filename( )
vr_recfl( )
vs_clip( )
vsf_color( )
vsf_interior( )
vsf_perimeter( )
vsf_style( )
vsl_color( )
vsl_ends( )
vsl_type( )
vsl_width( )
vsm_color( )
vsm_height( )

vsm_type( )

vst_alignment( )

vst_color( )

vst_effects( )

vst_font( )

vst_height( )

vst_load_fonts( )

vst_point( )

vst_unload_fonts( )

vswr_mode( )

Index

SYBEX

ATARI ST AND IBM PC SYSTEMS

PROGRAMMER'S GUIDE TO GEM

The **Programmer's Guide to GEM** is a detailed introduction and reference guide for anyone wishing to develop applications for the GEM operating environment. If you own, use, or program any computer using GEM—from the Atari ST series to the IBM PC—this is one text you shouldn't be without.

You'll gain a solid grasp of the tools and concepts of GEM programming through detailed, function-by-function discussions and clear illustrations. Step-by-step programming examples will show you how to put this knowledge to work. You will design and build a full-featured GEM application—a simple doodling program— using objects, events, windows, menus, alert boxes, and more. Throughout the text, the authors stress concepts and techniques for high-quality GEM programming that are of value to any applications developer:

- human factors in designing the user interface
- programming for presentation-quality graphics
- programming the VDI for truly portable graphics software
- using the Resource Construction Set for prototyping and easy software modification

Detailed appendixes make this the most complete and easy-to-use GEM reference guide for programmers, with

- a detailed GEM glossary
- a complete guide to VDI and AES functions
- a tutorial on the Resource Construction Set
- complete code for the sample application

All programming examples are in C.

About the Authors

Phillip Balma is a former project manager, writer, and technical consultant for Digital Research, Inc., the creators of GEM. He is now director of new product development at Sony Corporation. William Fitler was also a project manager and senior software engineer at Digital Research, and worked on the early design and prototyping of pre-GEM user interfaces. He is now manager of applications product development at Sony.

SYBEX books bring you skills—not just information. As computer experts, educators, and publishing professionals, we care—and it shows. You can trust the SYBEX label of excellence.

ISBN 0-89588-297-3

SYBEX GERMANY