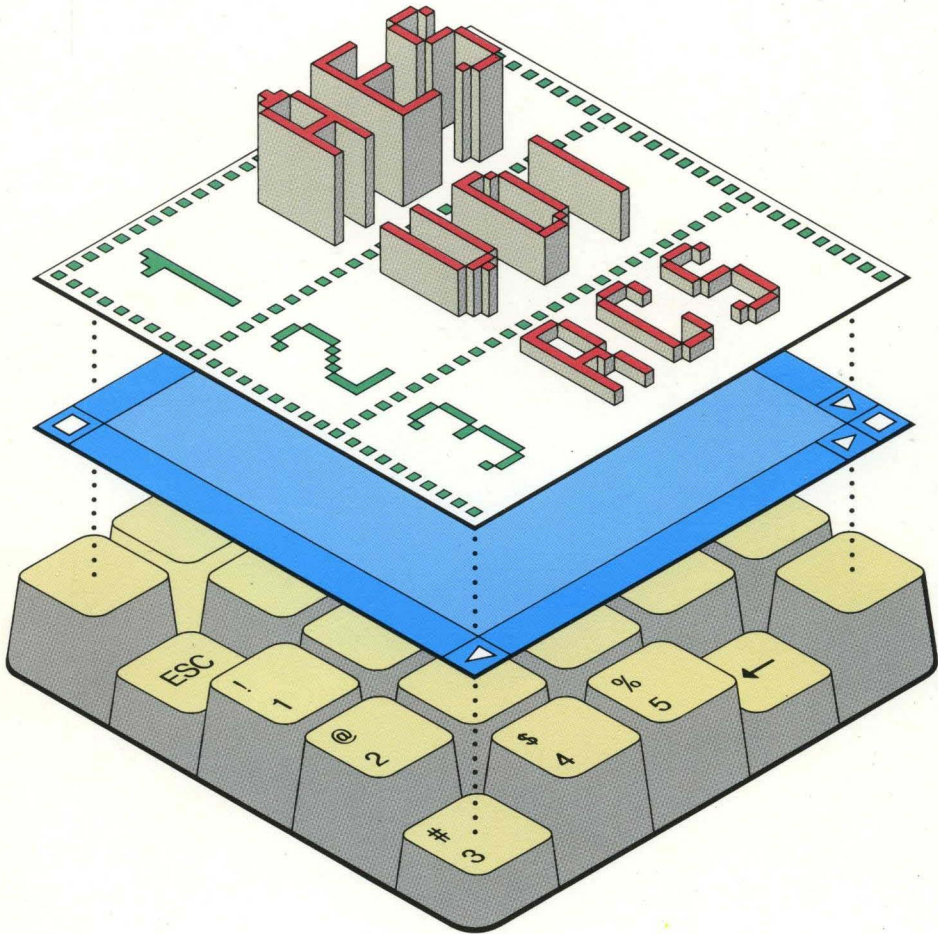


GEM *Programmer's Toolkit*™



**Introduction to
GEM™ Programming**

**Introduction to
GEM™ Programming**

COPYRIGHT

Copyright © 1986 Digital Research Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of Digital Research Inc., 60 Garden Court, P.O. Box DRI, Monterey, California 93942.

DISCLAIMER

DIGITAL RESEARCH INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Digital Research Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research Inc. to notify any person of such revision or changes.

NOTICE TO USER

This manual should not be construed as any representation or warranty with respect to the software named herein. Occasionally changes or variations exist in the software that are not reflected in the manual. Generally, if such changes or variations are known to exist and to affect the product significantly, a release note or READ.ME file accompanies the manual and distribution disk(s). In that event, be sure to read the release note or READ.ME file before using the product.

TRADEMARKS

Copyright © 1986. Digital Research and its logo are registered trademarks of Digital Research Inc. Graphics Environment Manager, GEM and its logo, GEM Desktop, Concurrent, GEM Paint, and GEM Draw are trademarks of Digital Research. Intel is a registered trademark of Intel Corporation. Motorola is a registered trademark of Motorola Incorporated.

Second Edition: June, 1986

Foreword

The GEM™ (Graphics Environment Manager™) programming environment provides two sets of functions for controlling graphics device output and pointing device and keyboard input. You access the functions through calls to the GEM entry point. The GEM Developer Kit includes C-language bindings to facilitate calling the functions and to help you develop portable programs. The kit also provides a GEM resource construction program, a series of C-language include files, and a symbolic instruction debugger.

GEM Documentation Set

- Introduction to GEM Programming: An overview of the GEM function sets--the Application Environment Services and Virtual Device Interface--with program examples demonstrating their use.
- GEM Desktop™ manual: The end user's guide to GEM Desktop features and operation.
- GEM Virtual Device Interface Reference Guide: The description of the GEM Virtual Device Interface (VDI) functions. This book is referred to as the VDI Reference Guide.
- GEM Application Environment Services Reference Guide: The description of the GEM Application Environment Services (AES) functions. This book is referred to as the AES Reference Guide.
- GEM Programmer's Utilities Guide: The description of the GEM RCS resource construction program, the symbolic instruction debugger, and the file transfer program provided in the GEM Developer Kit. Different versions of this book are provided for systems based on different microprocessors.

An understanding and appreciation of the GEM programming environment is best learned through use. Begin your introduction with the GEM Desktop manual, which provides GEM installation instructions, tutorials, and descriptions of the display features and input device handling.

When you are familiar with GEM operation, read the Introduction to GEM Programming. This book reviews the GEM interface characteristics, describes the system components, and gives program examples you can use in your program. Before proceeding with application development, read the GEM RCS description in the GEM Programmer's Utilities Guide. GEM RCS is a sophisticated tool that can save hours in program development time.

The VDI Reference Guide and AES Reference Guide contain the function descriptions. Refer to them during program development for argument and return value descriptions. In addition to the function descriptions, the VDI Reference Guide also provides the VDI error messages, metafile file format and reserved subcodes, standard keyboard values, character sets and font files, and bit image file format.

Intended Audience

This book is intended for experienced programmers. Formal education in graphics programming is helpful but not required to develop GEM applications and accessories. Familiarity with the C programming language is also helpful but not required.

Contents

This book has three sections.

Section 1 describes the GEM graphics features and summarizes the AES and VDI functions.

Section 2 provides background information that can affect application development.

Section 3 describes common GEM programming tasks and describes how to perform them with the AES and VDI functions.

Before proceeding, copy the disks provided in the GEM Developer Kit on your system and install the GEM RCS application. Store the original disks in a safe place.

Contents

1 GEM Graphics Features and Functions

1.1 AES	1-2
1.1.1 Menu Bar	1-2
1.1.2 Desktop Window	1-3
1.1.3 Application Windows	1-4
1.1.4 Dialog Boxes and Forms	1-8
1.1.5 Objects and Object Trees	1-10
1.1.6 User Input and AES Events	1-13
1.1.7 AES Function Libraries	1-13
1.2 VDI	1-17
1.2.1 Workstation Control Functions	1-18
1.2.2 Output Functions	1-18
1.2.3 Attribute Functions	1-19
1.2.4 Raster Operation Functions	1-20
1.2.5 Input Functions	1-20
1.2.6 Inquiry Functions	1-20
1.2.7 Escape Functions	1-20

2 GEM Components and System Interface

2.1 System and Device Characteristics	2-1
2.2 GEM Components	2-2
2.2.1 AES Components	2-2
2.2.2 Desk Accessories	2-4
2.2.3 Application Space	2-5
2.2.4 VDI Components	2-5
2.3 Component Relationships	2-6
2.4 Normalized Device and Raster Coordinate Systems	2-7
2.5 Porting Applications to Different Environments	2-9
2.6 Reserved Files	2-10
2.7 Enabling Graphics	2-12

3 Application Programming Tasks and Examples

3.1 Application Planning	3-1
3.2 DEMO Overview	3-2
3.3 Program Initialization	3-3
3.4 Event Management	3-9
3.4.1 Button Handling	3-12
3.4.2 Mouse Handling	3-15
3.4.3 Message Handling	3-16
3.4.4 Keyboard Handling and Text Output	3-18
3.5 Menu Processing	3-20
3.6 Form Processing	3-22
3.7 Work Area Maintenance	3-30
3.8 Program Termination	3-41
3.9 Creating Accessories	3-42

Tables

1-1 Window Control Areas	1-6
1-2 AES Libraries and Function Summaries	1-14
2-1 Reserved GEM Files	2-11

Figures

1-1 Menu Bar, Drop-down Menu, and Desktop Window	1-3
1-2 Application Window Components	1-5
1-3 Relationship of View Area to World Coordinate Space	1-8
1-4 Dialog Box as Form	1-9
1-5 Dialog Box and Panel	1-10
1-6 Raw Object Form of Figure 1-4	1-11
1-7 Object Tree Pointers	1-12
1-8 VDI Output Functions	1-19
2-1 GEM Components in System Memory	2-3
2-2 GEM Component Relationships	2-6
2-3 Normalized Device Versus Raster Coordinates	2-8
2-4 Aspect Ratio Conversion	2-9
3-1 OBJECT Structure	3-23
3-2 DEMO Pen/Eraser Selection Dialog Box	3-25
3-3 Window Rectangles	3-33

Listings

3-1	Opening Virtual Workstation.	3-4
3-2	Transforming Objects and Initializing APPLBLK Structures	3-6
3-3	Object Transformation.	3-7
3-4	Set-up Message Buffer and Screen	3-9
3-5	Main Event Loop	3-11
3-6	Button Handling.	3-13
3-7	Mouse Handling.	3-15
3-8	Message Handling	3-17
3-9	Checking for CTRL-C and Setting Text Attributes.	3-19
3-10	Character Output	3-20
3-11	Menu Handling.	3-21
3-12	Setting the Dialog Box's Current Selections	3-26
3-13	Display and Processing of a Dialog Box	3-27
3-14	Getting Data and Resetting SELECTED Flag	3-28
3-15	Save Work Area	3-31
3-16	Redrawing a Portion of the Screen	3-34
3-17	WM_SIZED and WM_MOVED Message Responses	3-36
3-18	WM_FULLED Message Response.	3-37
3-19	Arrow and Slider Message Responses	3-39
3-20	Updating the undo_mfdb, Sliders, and Screen.	3-40
3-21	Program Termination Routine.	3-42
3-22	HELLO Event Handler	3-45

GEM Graphics Features and Functions

The GEM programming environment provides applications with graphics features such as drop-down menus, windows, icons, and dialog boxes for managing the user interface. The program interface has two components:

- **Application Environment Services (AES):** Sets of functions that manage screen output and mouse and keyboard input.
- **Virtual Device Interface (VDI):** An extensive collection of device-independent functions that manage the interface to the physical graphics devices.

The GEM components do not replace the computer's operating system; they supplement the operating system's functions with the AES and VDI functions. Applications use the operating system for file management and the GEM functions to manage user input and graphics device output.

The GEM software runs on computers from many manufacturers and under several operating systems. With careful use of the AES and VDI functions in languages such as C or Pascal, you can produce programs that are portable between systems supporting the GEM environment.

Note: The AES supports a variety of pointing devices. For convenience, "mouse" is used in this manual to refer to all types of pointing devices. The on-screen representation of the pointing device's location is called the mouse form.

1.1 AES

The AES is composed of twelve function libraries. (The term "library" in this context means, "a set of related functions.") Each library addresses a different aspect of the AES programming interface. For example, Form Library functions attend to the display and processing of dialog boxes and other types of user-interactive forms; Window Library functions control the creation, display, and inquiry of windows; and Event Library functions monitor user input, message input, and timer events. Section 1.1.7 lists the libraries and the functions in each.

The AES interface to the screen has two fundamental components: the menu bar and the desktop window. Figure 1-1 illustrates the relative positions of the menu bar and desktop window on the screen. This figure also shows how a drop-down menu is displayed on the desktop window portion of the screen.

Note: The expression "desktop window" refers to the window created by the AES that serves as the backdrop for its windowing system. Do not confuse the desktop window with the GEM Desktop. The latter is an application that creates its own windows on the desktop window.

1.1.1 Menu Bar

The menu bar is a reserved area of the screen managed by the AES. The entries in the menu bar, for example, Files, Options, and Accs in Figure 1-1, are called titles. The boxes displayed below titles are called drop-down menus. The entries in drop-down menus are called items. The title on the far right (Accs in the figure) is always the desk accessory menu. (See Section 2.2.2 for the description of desk accessories.)

The AES manages mouse and button input and the window display while the mouse form is in the menu bar. As soon as the mouse form enters the menu bar, the AES converts it to an oblique arrow. When the user moves the mouse over a title, the AES converts the title to reverse video and displays the title's drop-down menu. After the user clicks on a selection, the AES redraws the screen but leaves the title in reverse video. It is the application's responsibility to return the mouse form to its application-defined form and convert the title back to normal video in its menu handling routine.

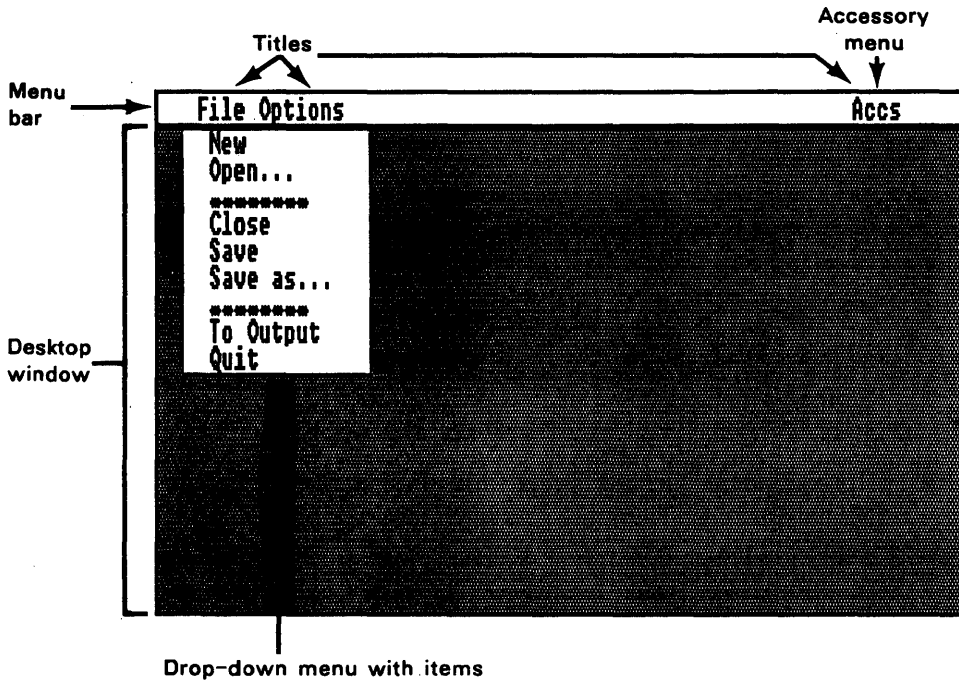


Figure 1-1. Menu Bar, Drop-down Menu, and Desktop Window

The menu bar is blank until you provide the contents. You use Menu Library functions to control the contents. The functions allow you to display and remove a set of titles and drop-down menus, enable and disable titles and items, check selected items, register and unregister desk accessories in the accessory menu, and change menu item text. You construct sets of titles and their drop-down menus with the GEM RCS application.

1.1.2 Desktop Window

The desktop window is the portion of the display surface below the menu bar. This area is the total space available to you for display output. You can output directly to the desktop window, however, the standard practice is to create an application window.

All windows are identified by a unique value called the handle. The AES manages window handle assignments. The desktop window handle is always zero.

You get the size of the desktop window with an AES Window Library function. The function returns the width and height as the number of pixels (picture elements) along the window's x and y axes. The desktop window's coordinate addressing system has its origin point ($x = 0$ and $y = 0$) in the upper lefthand corner. You reference individual pixels by their x and y coordinates relative to this point. See Section 2.4 for more about coordinate systems.

1.1.3 Application Windows

An application window is the portion of the desktop window reserved by the program for its use. You can make an application window as big as the desktop window, however, an application window cannot be larger. You create and manage application windows with AES Window Library functions. You can create multiple application windows and they can overlap.

The AES manages up to eight windows. Because the desktop window is always present and cannot be deleted, seven windows are available to the application. Be careful in your window use, however. If your application creates seven windows, no desk accessories (which each require their own window) can be run until a window has been deleted. Note that closing a window does not delete it.

Application Window Components

An application window is composed of selectable border components and a work area. You select the border components and specify the window area and location when you create the window. The AES displays the border components selected within the window area requested. The space remaining becomes the application window's work area. Figure 1-2 shows an application window with all border components.

The border components fall into two groups: informational and control. The informational components are a title bar and an information line. The title bar is an area one character-cell high displayed across the top of the window. The information line appears right below the title bar and is also one character-cell high. You output the title bar and information line contents with Window Library functions. The longest possible string is 80 characters for each. However, the actual number available depends on the window width and character size.

The window control areas are individually selectable boxes, bars, and scroll bars with sliders that give the user the ability to manipulate the application window's size, location, and view area. (The view area is described later on in this section.) Table 1-1 lists the window control areas available.

Note: The AES does not require you to respond in a specific fashion when the user clicks on a window control area. The responses listed in Table 1-1 are recommended responses consistent with the responses in other GEM applications.

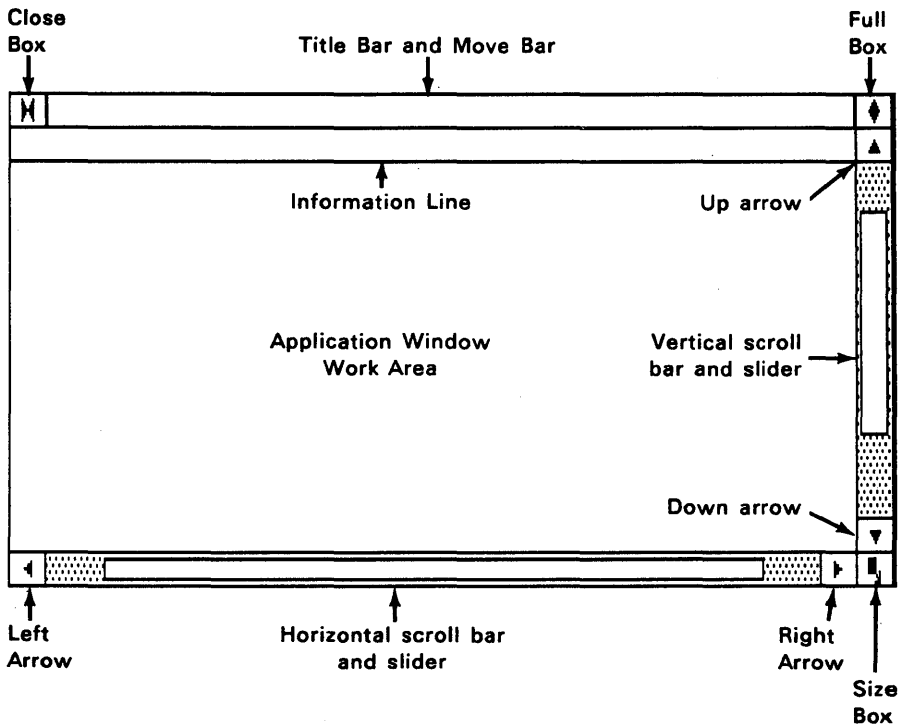


Figure 1-2. Application Window Components

Table 1-1. Window Control Areas

Component	Program Response
Arrows	(▼ ▲ ◀ ▶) Change the window's view area by one program-defined increment.
Close box	Close the window or perform a concluding action consistent with the window's function.
Full box	Toggle window between present size and greatest possible size.
Move bar	Relocate the window. Note: There is no distinct move bar. This feature is given to the title bar when the move bar attribute is selected. You must give the title bar a name for the move bar function to work.
Scroll bar and slider	Change the window's view area horizontally and/or vertically by a program-defined page or to the user-selected location.
Size box	Change the size of the window to the user-selected height and width.

The window control areas, like the menu bar, are reserved areas in which the AES monitors button and mouse input. Unlike the menu bar, however, the AES does not change the mouse form when it enters the control areas. If you want the mouse form to change when it enters the window control areas, the application must change it.

Work Area

The application window's work area is the rectangle enclosed by the window control areas. The size and location of the work area can vary during program operation depending upon the user's use of the size and move boxes. You get the location and dimensions of the work area with the Window Library's `wind_get` function.

The application window work area defines the area in which the application controls user input and screen output. You manage user input with the AES Event and Form Library functions. You manage the screen output with the AES Window, Object, Graphics, and Extended Graphics Library functions and the VDI output and attribute functions.

View Area

The view area is the portion of the world coordinate space shown in the application window's work area. The world coordinate space is the application's total addressable window data area. The user controls the portion of the world coordinate space shown in the view area with the arrows, scroll bars, and sliders. If you do select these window control areas the user cannot change the view area.

The size of the view area changes with the size of the application window work area and any scaling capability provided in the application. If you do not specify a size box, the user cannot change the size of the application window work area.

The scroll bar and slider provide the visual representation of the view area location and size with respect to the world coordinate space. Figure 1-3 illustrates these components and their relationship. The AES controls the actual display of the scroll bar and slider, however, the application must tell the AES the slider's size and location.

You manage the slider size and location with Window Library `wind_set` function. To set the slider size, you designate the proportion of the scroll bar filled by slider as a value between 1 and 1000 where 1 specifies the smallest slider size and 1000 specifies a slider that fills the scroll bar. To set the horizontal and vertical slider locations, you designate the position of the horizontal slider's left edge or the vertical slider's top edge relative to the scroll bar where 1 is the left/top of the scroll bar and 1000 is the right/bottom.

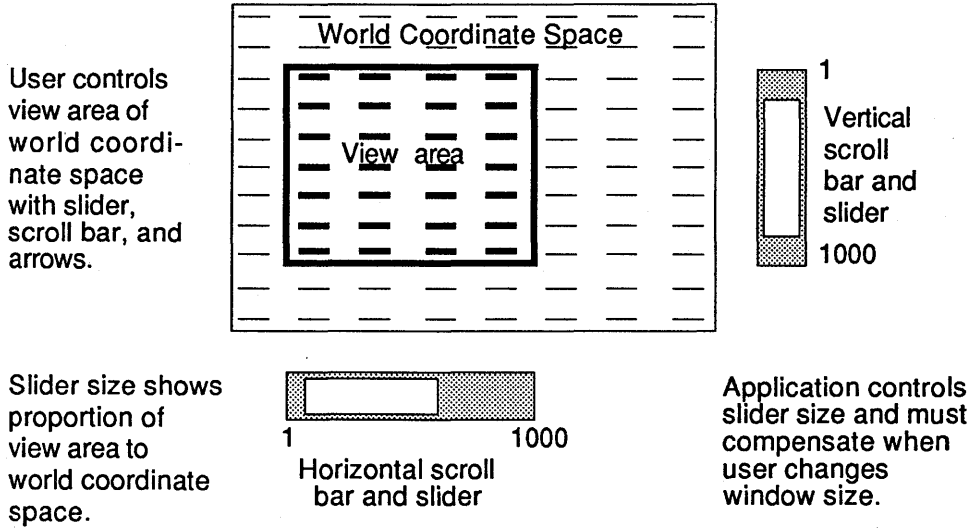


Figure 1-3. Relationship of View Area to World Coordinate Space

1.1.4 Dialog Boxes and Forms

The AES Form and Object Libraries provide functions for handling dialog boxes. Dialog boxes are rectangles within which you can mix different types of text- and graphics-objects in any combination and format. The rectangle can be any size up to the full desktop window. There are two general types of dialog boxes: transient displays that collect specific information and on-going fixtures of the application window.

Transient dialog boxes, called forms, are distinguished by two characteristics: they limit user interaction to the form and they have exit buttons. To the user, the program enters a mode when a form is displayed. Menu bar and window control options are not available until the user selects an exit button. Figure 1-4 shows a form.

Dialog boxes that are on-going fixtures typically do not have the restrictions of the form. That is, the user is free to move into and out of the dialog box without selecting an exit button to leave. In comparison to the form, this type of dialog box is modeless. Figure 1-5 shows two modeless dialog boxes.

Note: Both of the following figures are taken from the GEM RCS application. See the GEM Programmer's Utilities Guide for the description these dialog boxes and panels.

LOCKED, NORMAL, and EXPERT options demonstrate radio button flag attribute. If user selects LOCKED, NORMAL is deselected.

OK and Cancel are exit buttons. User must click on one or the other to leave the form.

Resource File Editing Protection:

LOCKED Objects may be edited, sized, or moved, but the object tree structure may not be changed.

NORMAL A warning is given before the workspace is cleared or trees are rearranged.

EXPERT No warnings are given when trees are altered or the workspace is cleared.

Figure 1-4. Dialog Box as Form

The dialog box above contains three options and two exit buttons. The three options have the radio button flag attribute. This means that only one option can be selected at a time. Other general types of options are check boxes, where the user can select any combination of options, and editable text boxes.

Typically, you remove the form when the user is through with it. Unlike drop-down menus, the AES does not automatically redraw the screen when the user selects an exit button; it is the application's responsibility to remove the dialog box.

Figure 1-5 shows a dialog box along the left side of the figure and a variation of the dialog box called a panel along the bottom. Dialog boxes and panels are functionally similar, however, panels give you a bit more flexibility for arranging the objects within the box.

The dialog box below is composed largely of bit image objects. This type of object and the icon type are the most complex to create but gives you the ability to set the object's appearance pixel-by-pixel. The other object types, such as graphics boxes, graphics text, and buttons,

have specific shapes and contents. The panel along the bottom of the figure illustrates all of the object types available for constructing dialog boxes and panels.

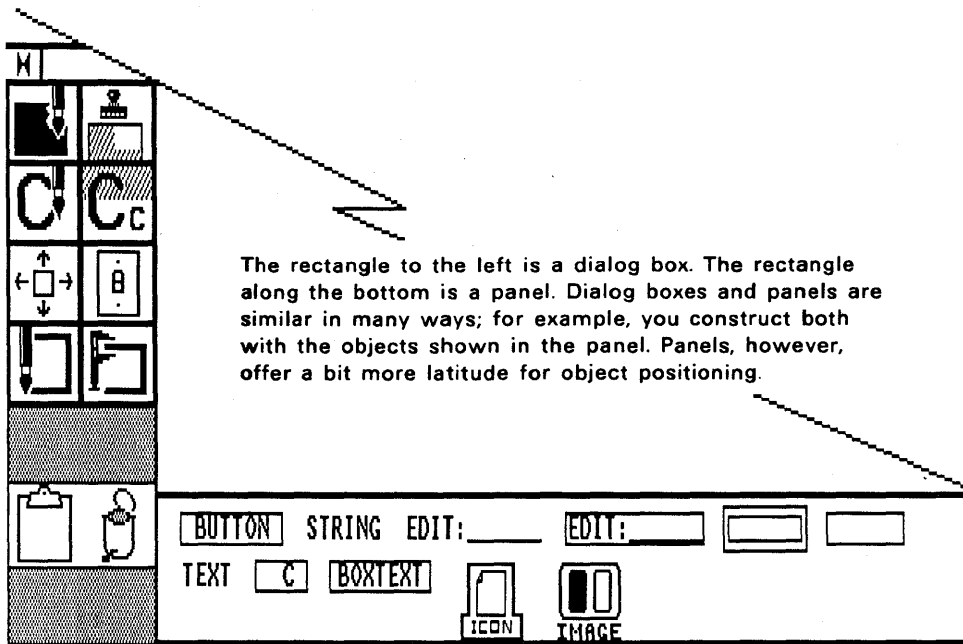


Figure 1-5. Dialog Box and Panel

1.1.5 Objects and Object Trees

You build dialog boxes, panels, menu bar titles and drop-down menus, and special dialog boxes called alert boxes with objects. An object is a structure describing the contents of a rectangular portion of the screen. Figure 1-6 below shows the dialog box in Figure 1-4 in its raw object form. The objects appear in the figure as they do before the object text is edited.

Every object has an OBJECT structure. The OBJECT structure defines the object's size, location, flags, state, and other variables. Figure 3-1 illustrates the OBJECT structure format. See the description of the

Object Library functions in the AES Reference Guide for the description of the object attribute flags and object states.

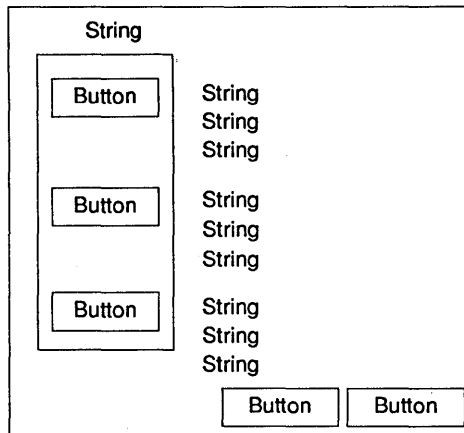


Figure 1-6. Raw Object Form of Figure 1-4

The objects in a menu bar and its drop-down menus, dialog box, panel, and alert box are linked in an array of OBJECT structures called an object tree. Each tree has a starting object called the root. The remaining objects are linked through three types of pointers maintained in each OBJECT structure. The types are defined as follows:

- the object's first child--a head object
- the object's last child--a tail object
- the object's next sibling or, if there's no sibling, its parent

The pointer is the index of the object's OBJECT structure relative to the tree's root object. Figure 1-7 shows the object pointers of the dialog box shown in Figure 1-4.

You use GEM RCS to create object trees. GEM RCS records the trees in a resource file. The Resource Library provides functions for loading a resource file and getting the address of a tree's root object. You use the root object address to display the tree and process user interaction with Object and Form Library functions.

The use of object trees is an important aspect of GEM application programming. The Object Library description in the AES Reference Guide explains the OBJECT and related data structures. Section 3.6 contains demonstrations of Object and Form Library function use for the display and processing of object trees.

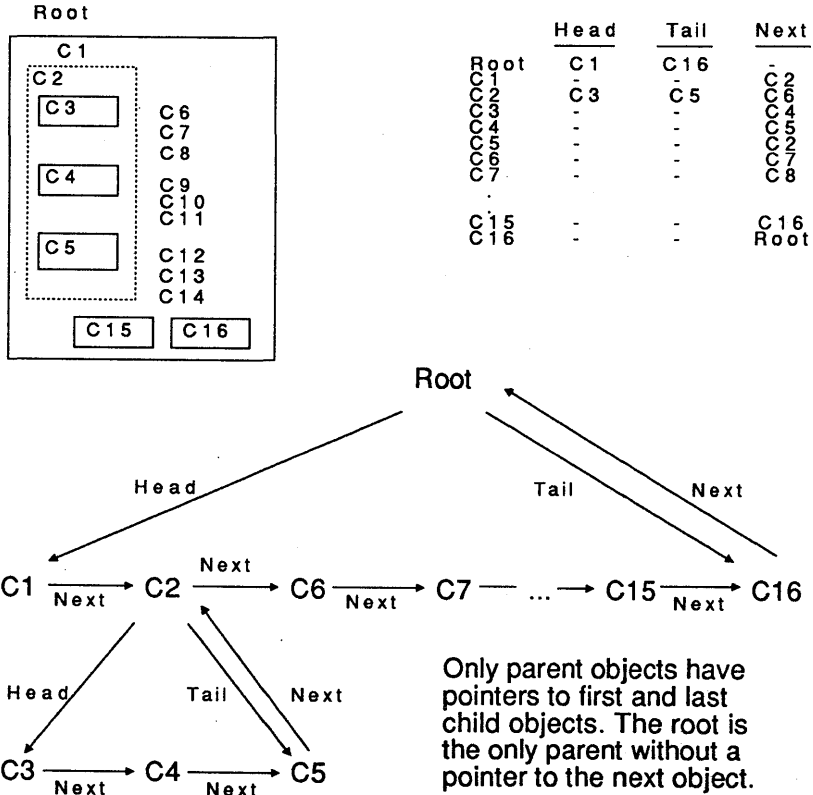


Figure 1-7. Object Tree Pointers

1.1.6 User Input and AES Events

The AES provides functions for monitoring three input devices: the keyboard, a mouse, and mouse buttons. The AES reports user input from these devices through an event and message system.

Note: The VDI includes a set of input functions. However, use the AES input functions only for all GEM applications.

You use the AES Event Library to manage the event and message system. The library supports the following input events:

- one or more state changes on one or more mouse buttons
- keyboard entry
- mouse form movement into or out of a designated region
- receipt of a message
- expiration of a time interval

The AES uses the message system to report input events occurring while the mouse form is outside the application window's work area and application window status changes. Each message indicates the event that occurred and provides related data.

Note: You can also use the message system to transfer data between different applications, between an application and an accessory, and from an application to itself. In these cases, you use Application Library functions to read and write the message.

1.1.7 AES Function Libraries

There are twelve AES function libraries. Each one provides services for managing a different aspect of the graphics environment. Table 1-2 lists the functions in each library.

Table 1-2. AES Libraries and Function Summaries

Application Library: Application initialization and intercommunication

<code>appl_bvset</code>	Set the disk configuration.
<code>appl_exit</code>	End application execution and release memory.
<code>appl_find</code>	Return identification number of another application.
<code>appl_init</code>	Establish application's internal data structures and return application's identification number.
<code>appl_read</code>	Read from a message pipe.
<code>appl_tplay</code>	Play recording of user input.
<code>appl_trecord</code>	Record user input.
<code>appl_write</code>	Write to a message pipe.
<code>appl_yield</code>	Force a dispatch.

Event Library: Input and event management

<code>evnt_button</code>	Wait for button input.
<code>evnt_dclick</code>	Set/get double-click time period.
<code>evnt_keybd</code>	Wait for keyboard input.
<code>evnt_mesag</code>	Wait for a message.
<code>evnt_mouse</code>	Wait for the mouse form to enter or leave a region.
<code>evnt_multi</code>	Wait for any of the above events to occur.
<code>evnt_timer</code>	Wait for a time interval to pass.

Menu Library: Menu bar display and contents management

<code>menu_bar</code>	Display or remove application's menu bar.
<code>menu_ichck</code>	Display or erase check next to an item.
<code>menu_ienable</code>	Display item in full or reduced intensity.
<code>menu_register</code>	Add text string to desk accessory menu.
<code>menu_text</code>	Change text in an item.
<code>menu_tnormal</code>	Display title in normal or reverse video.
<code>menu_unregister</code>	Remove text string from desk accessory menu.

Table 1-2. Continued

Object Library: Object tree display and object management

<code>objc_add</code>	Add an object to a tree.
<code>objc_change</code>	Change object state.
<code>objc_delete</code>	Delete an object from a tree.
<code>objc_draw</code>	Draw an object tree.
<code>objc_edit</code>	Edit text in an object.
<code>objc_find</code>	Return index of object at current mouse form location.
<code>objc_offset</code>	Return object location relative to screen.
<code>objc_order</code>	Change object order in a tree.

Form Library: Form display and input management

<code>form_alert</code>	Display an alert box.
<code>form_button</code>	Process button input.
<code>form_center</code>	Center a dialog box in the work area.
<code>form_dial</code>	Reserve or free work area space for a dialog box.
<code>form_do</code>	Process a user-interactive form.
<code>form_error</code>	Display an error box.
<code>form_keybd</code>	Process keyboard input.

Graphics Library: Rectangle draw and move

<code>graf_dragbox</code>	Draw rectangle that moves with mouse input.
<code>graf_handle</code>	Get workstation handle assigned by VDI.
<code>graf_mbox</code>	Draw rectangle that moves from one location to another.
<code>graf_mkstate</code>	Return mouse location and button state and keyboard state.
<code>graf_mouse</code>	Change mouse form.
<code>graf_rubbox</code>	Draw rectangle that expands or contracts with mouse input.
<code>graf_slidebox</code>	Move rectangle within another rectangle according to mouse input.
<code>graf_watchbox</code>	Change object state when mouse form moves into or out of a rectangle.

Table 1-2. Continued

Window Library: Window management

wind_calc	Calculate size of window with or without control areas.
wind_close	Close window.
wind_create	Create window and return window handle.
wind_delete	Delete window.
wind_find	Return handle of window underneath mouse form.
wind_get	Get window information.
wind_open	Open window.
wind_set	Set window characteristics.
wind_update	Notify AES that a window update is to begin or end or mouse control by program is to begin or end.

Resource Library: Resource file and object address management

rsrc_free	Release memory allocated to a resource file.
rsrc_gaddr	Get address of an object or object tree.
rsrc_load	Load a resource file of object trees.
rsrc_obfix	Convert object location and size to pixel coordinates.
rsrc_saddr	Store address of an object or object tree.

Scrap Library: Temporary file management

scrp_clear	Erase all SCRAP files in the current scrap directory.
scrp_read	Get current scrap directory path.
scrp_write	Set scrap directory path.

File Selector Library: Directory display and file selection

fsel_input	Process file selection dialog.
------------	--------------------------------

Table 1-2. Continued

Shell Library: Shell information retrieval and management

<code>shel_envrn</code>	Search for an environment parameter string.
<code>shel_find</code>	Search for a file and return directory path.
<code>shel_rdef</code>	Return default application's command and directory path.
<code>shel_read</code>	Return application's command line (including tail).
<code>shel_wdef</code>	Set new default application command and directory path.
<code>shel_write</code>	Set command to be performed when application terminates.

Extended Graphics Library: Rectangle expand and shrink

<code>xgrf_stepcalc</code>	Calculate increments for expanding or contracting rectangle.
<code>xgrf_2box</code>	Draw series of expanding or contracting rectangles.

1.2 VDI

The VDI provides device-independent functions for opening and closing, setting attributes for, drawing on, and getting information from graphics devices. Input functions are also provided; however, they should not be used in the same program with the AES event functions.

The VDI functions complement the AES functions--each serving a different purpose. Use the AES calls to initialize the application, manage menus and window control areas, and perform object-based operations within the work area. Use the VDI functions to open and initialize each graphics device and to output graphics and text.

The VDI functions, like the AES functions, can be grouped according to use. Space precludes listing all the function definitions here. The remainder of this section describes the groups.

1.2.1 Workstation Control Functions

A workstation is a generic term for any graphics device. Common graphics devices are a screen, a mouse, and a keyboard, a graphics printer, and a plotter. Although it is not a graphics device, you treat metafiles in much the same ways as a graphics device. (Metafiles are recorded versions of an image; see Section 2.6 for the description of metafiles.)

Each graphics device has two sets of attributes. One set consists of output attributes, such as color, line style, and text face, that you can set dynamically during program execution. The other set consists of device characteristics, such as maximum addressable width and height, pixel size, minimum line width, and maximum number of colors, that cannot be changed at runtime.

You use the Open Workstation workstation control function to open graphics devices and get the device identifier, called the handle. You use the handle to specify the device in all subsequent access calls. You use the Open Virtual Workstation function to open the display screen. This gives the application a virtual screen workstation with a set of output attributes independent of the physical screen's set.

In the Open Workstation call, you specify the driver to load, set the coordinate system, and set the default values for the output attributes. The Open Virtual Workstation call is the same except that you use the handle to specify the physical device. Both calls return an array with the device-dependent characteristics and the handle.

The remainder of the workstation control functions are used to load and unload fonts, clear and update the workstation display surface, and enable and disable rectangle clipping. (Clipping limits the area in which graphics output is displayed--see Section 3.4.1 for examples of rectangle clipping.)

1.2.2 Output Functions

The output functions provide calls for drawing lines between two or more points, a marker at one or more points, text, filled areas, and so forth. Also provided are a set of generalized drawing primitives for bars, circles, circular arcs and pie slices, ellipses, elliptical arcs and pie slices, rounded rectangles (hollow and filled), and justified text. Figure 1-8 illustrates the output functions available.











Polyline		Pie slice	
Polymarker	*****	Rectangle	
Graphic text	hello	Ellipse	
Bar		Elliptical arc	
Circle		Elliptical pie slice	
Arc		Rounded rectangle	

Figure 1-8. VDI Output Functions

Each output function has a set of attributes associated with it. For example, bars have the fill interior style, fill style index, writing mode, fill color, and fill perimeter style attributes while lines have a writing mode, line type, width, color, and end style. You set the attributes independent of the functions with VDI attribute function calls; you do not set attributes in the output function call. Note, however, that the Open Workstation allows you to set default values for many attributes.

1.2.3 Attribute Functions

The attribute functions control the writing mode (how the item drawn is imposed over the existing pixel values) and the individual characteristics for the output functions (line, marker, character, text, and fill). Once set, an attribute remains set until you change it or the program terminates.

1.2.4 Raster Operation Functions

Raster operations apply to pixels and rectangular blocks of pixels. The functions copy blocks, set the copy mode (like writing mode, copy mode determines how the pixel block is imposed over the existing pixel values), and transform a block between standard and device-specific formats.

1.2.5 Input Functions

The input functions return the input from locator (mouse, trackball, or joystick), valuator (potentiometer), choice (function keys), and string (keyboard) devices. Two access modes are provided for each device: request and sample. In sample mode, the call returns immediately with device status and any data available. In request mode, the function waits until data is available.

Additional functions are provided for the following purposes:

- set the mouse form
- show or hide the mouse form
- exchange timer interrupt, button change, mouse movement, and mouse form change vectors
- sample the mouse button and keyboard state

1.2.6 Inquiry Functions

The inquiry functions return the current attribute values for the different output functions, fonts loaded, input mode, and cell size. All attributes are returned from a single call. For example, the Inquire Fill Area function returns an array with the values for the fill area interior style, color index, style index, writing mode, and perimeter style.

1.2.7 Escape Functions

The escape functions access the special capabilities of a graphics device. The VDI predefines some escape functions, such as those for controlling the cursor on a screen in alpha mode and managing metafile output. The driver writer can also define escape functions.

End of Section 1

GEM Components and System Interface

This section describes the following:

- Computer system graphics drivers and their characteristics.
- GEM components and their interface to the application, operating system, and hardware.
- Raster and Normalized Device Coordinates.
- Porting applications to different environments.
- The files used by GEM for initialization, accessory and object storage, and image recording.
- How to load GEM for different purposes.

2.1 System and Device Characteristics

GEM supports a variety of output devices through a set of device drivers provided to the user with GEM software. The user selects the drivers corresponding to his or her hardware configuration during GEM installation. The GEM installation software then builds an ASSIGN.SYS file that correlates each device with a device ID number and its font files. The application uses the device ID from the ASSIGN.SYS file in its Open Workstation call. See the VDI Reference Guide for the ASSIGN.SYS description.

The VDI provides a standard interface to all devices. However, device characteristics such as the display surface size, pixel dimensions, resolution, and supported functions vary from system to system and device to device on the same system. For example, the screen dimensions on one system can be 640 by 400 pixels while on another it is 320 by 200. Meanwhile, the resolution of a laser printer compatible with both systems might be 300 pixels by 300 pixels.

You get the device characteristics from the output array of the Open Workstation call or, in the case of the screen, the Open Virtual Workstation call. You find out from these calls, for example, the device's

- pixel aspect ratio (the ratio of pixel width to height)
- number of character heights, linetypes, linewidths, marker types, faces, patterns, and colors
- supported Generalized Drawing Primitives
- color, text rotation, area fill, and cell array capability

The VDI Extended Inquire function returns additional device-dependent information. It is up to you to interpret device data, ensure that the functions used in your program are supported by the device, and adjust output data to compensate for the device characteristics.

2.2 GEM Components

When GEM is loaded, it takes over management of the memory available for transient program execution. The significant system subdivisions when GEM is loaded are the operating system, AES, VDI, desk accessories, and application space. Figure 2-1 illustrates how system memory is allocated to these subdivisions.

2.2.1 AES Components

The AES consists of a kernel, the Screen Manager, the Dispatcher, and a menu and alert buffer. The kernel provides the AES function support. All functions except Event Library functions are processed immediately. The event functions cause a dispatch to let other processes, such as the Screen Manager or a desk accessory, run.

The Screen Manager is a background process that manages the screen display and button/keyboard input when the mouse form is in the following areas:

- menu bar and drop-down menus
- title bar
- information line
- window control areas

The Screen Manager reports user input made while the mouse form is in these areas to an application using predefined messages.

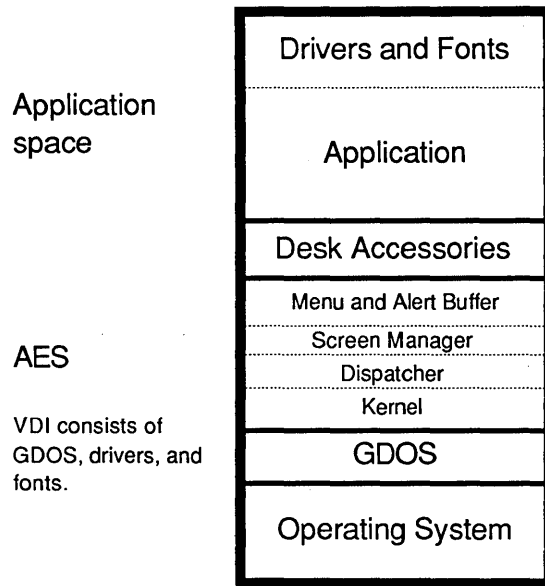


Figure 2-1. GEM Components in System Memory

The Screen Manager also provides automatic form processing through Form Library functions. The Screen Manager records user input in the form's object tree's editable objects and returns to the application when the user selects an exit button.

The Dispatcher allocates CPU time and maintains process ready and not-ready lists. Processes sharing CPU time are the application, the Screen Manager, and other background processes such as a desk accessory. A dispatch occurs on every tenth call to the AES or when a process calls an Event Library function, whichever comes first. You can force a dispatch with the `appl_yield` function.

The Dispatcher allocates time to processes on the ready list on a round-robin basis. A process remains on the ready list until it calls an Event Library function. The following events are supported:

- a keystroke
- mouse button entry
- mouse movement into or out of a specified rectangle
- an AES or interprocess message
- expiration of a time interval

Note: Applications that do not make many AES calls can dominate CPU time to the exclusion of other processes, including the Screen Manager. To avoid this condition, include occasional `appl_yield` calls in your code.

The process remains on the not-ready list until the event completes. At the next dispatch after event completion, the dispatcher adds the now-ready process to the ready list.

The menu and alert buffer is reserved memory used by the Screen Manager to hold the area of the screen overwritten by a drop-down menu or alert box. The size of this buffer is 1/4 the screen size. The amount of memory allocated to the buffer depends on screen resolution and the number of color planes. (Color planes are described in the VDI Reference Guide.)

2.2.2 Desk Accessories

Desk accessories are programs loaded during GEM initialization that remain in memory as long as GEM is resident. GEM loads up to six desk accessories from the the first three files with the ACC file extension in the GEMBOOT directory. Each file can contain more than one desk accessory. Note that loading stops if the addition of another desk accessory reduces the amount of memory left for application space to less than a minimum amount.

The desk accessory names are listed in the rightmost drop-down menu. The title of this menu is the name of the application running at the time. Each desk accessory must make a `menu_register` call to register its name in menu. To remove its name, the accessory uses the `menu_unregister` function.

Desk accessories and applications, though similar in many respects, differ in others. See Section 3.9 for the description of the differences between desk accessories and applications.

2.2.3 Application Space

The application space is memory reserved by GEM for applications and the drivers, fonts, and resource files they load. The minimum application space allowed by GEM 2.x is 192 kilobytes. (The minimum allowed by GEM 1.x is 128K.) The 192K minimum, however, is based on a monochrome screen with a resolution of 640 by 200. On systems with color screens or higher resolution, the minimum is less to accommodate the increased needs of the menu and alert buffer.

GEM automatically allocates and deallocates application space memory to drivers, fonts, and the resource file as the application makes Open and Close Workstation, Load and Unload Font, and `rsrc_load` (load a resource file) and `rsrc_free` (free memory allocated to the resource file) calls. However, the application must make operating system memory allocation calls to get temporary memory. If you need contiguous memory, be sure to make the open, load, and allocate calls before an Event Library call and within the first ten AES calls.

2.2.4 VDI Components

The VDI has three logical components: the Graphics Device Operating System (GDOS), graphics device drivers, and fonts. The GDOS controls VDI function processing and the assignment of device handles. The graphics device drivers are the interface to the physical devices. Fonts are character bit images loaded from file. Each type face has an individual font file.

The screen and mouse drivers are loaded as an integral part of the GEM system. The screen driver includes a system font. The screen driver is usually the only device that includes a default font. You can load additional screen fonts if more are listed in the ASSIGN.SYS file. The GDOS loads all drivers besides the screen and mouse drivers and all fonts besides the system font into the application space.

2.3 Component Relationships

Figure 2-2 illustrates the functional relationship between the application, desk accessories, GEM components, and the operating system. Applications and desk accessories use the functions of all three services--the AES for its menu, window, and input functions; the VDI for its device initialization and output functions; and the operating system for its file system functions.

Note that the AES goes through the VDI for screen output and user input. This provides AES function portability from one system to another.

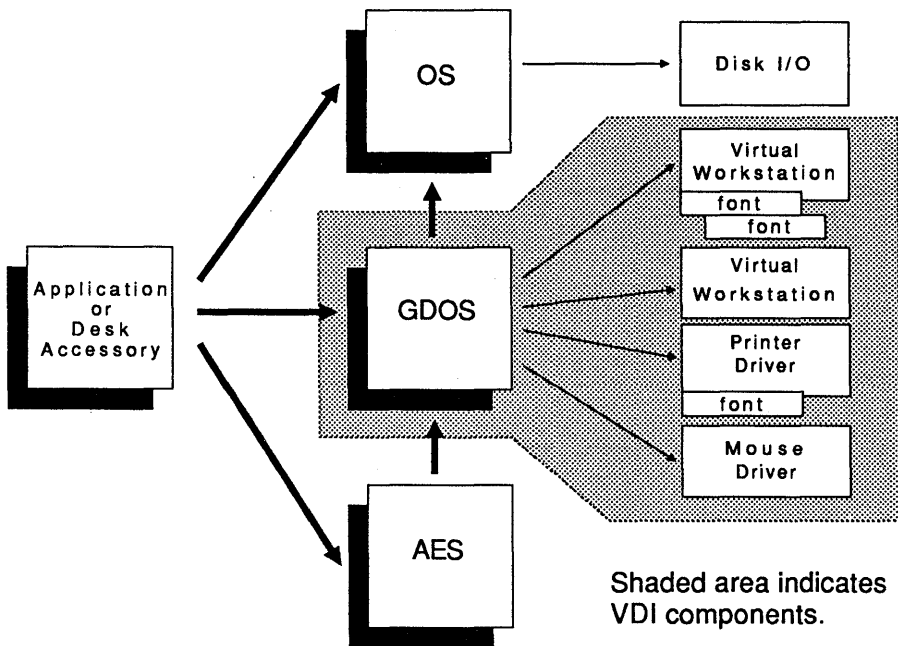


Figure 2-2. GEM Component Relationships

2.4 Normalized Device and Raster Coordinate Systems

All computer graphics are displayed using a coordinate system to reference individual points on the workstation. The VDI supports two coordinate systems: normalized device coordinates and raster coordinates. You specify which coordinate system you are going to use in the Open Workstation and Open Virtual Workstation calls.

Note: The AES requires you to use raster coordinates to reference points on the display device.

Normalized device coordinates (NDC) provide a standard addressing scheme independent of the number of actual picture elements (pixels) supported by the device. The address space defined by NDC is square with 32,768 units on each axis. A point referenced by NDC coordinates appears at the same relative location on the surface regardless of the device.

Raster coordinates (RC) provide an addressing scheme that references locations according to their pixel coordinates. The number of pixels on each axis is defined by the device driver. You get the number of pixels on each axis and the pixel dimensions (pixels are not necessarily square) from the Open Workstation or Open Virtual Workstation functions' output arrays.

An important difference between normalized device and raster coordinates is the origin point (the point where the x and y axes meet). For normalized device coordinates, the origin point is in the lower lefthand corner. For raster coordinates, the origin point is in the upper lefthand corner. Figure 2-3 illustrates the differences between normalized device and raster coordinates.

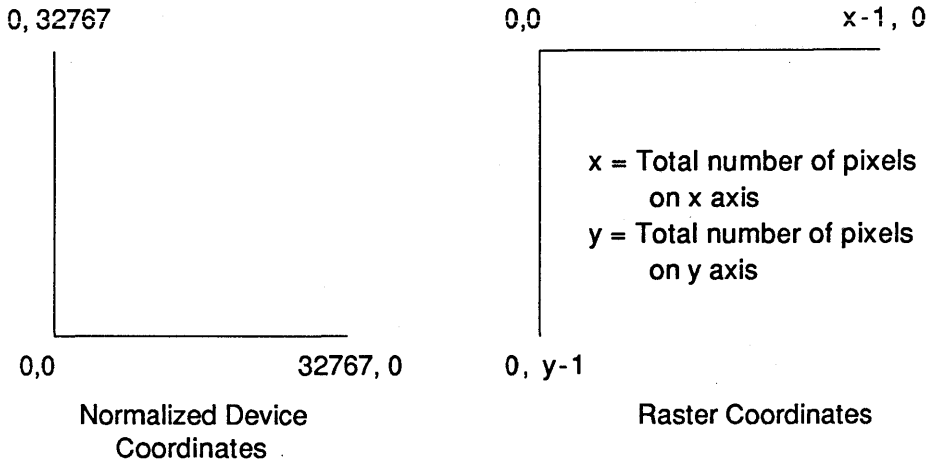


Figure 2-3. Normalized Device Versus Raster Coordinates

The VDI translates normalized device coordinates to raster coordinates before outputting point references to a device driver. Because the VDI maps the full NDC range (0 to 32,767) to each axis, the aspect ratio (the ratio of the horizontal to vertical dimensions) is not 1:1 for devices with an unequal number of pixels on the axes. Note that the pixel dimensions also affect the aspect ratio.

Figure 2-4 shows how the aspect ratio distorts a polyline square drawn in NDC coordinates when printed on a device with an address space 600 pixels by 400 pixels. The source is 16,383 units square (half of the total addressable space). In raster coordinates, the square is printed as a rectangle 300 by 200 pixels.

The VDI compensates for the aspect ratio when you use the generalized drawing primitives to draw circles, circular arcs, and circular pie slices. (The generalized drawing primitives are described in the VDI Reference Guide description of the output functions.) For all other output functions and generalized drawing primitives, you must compensate for the aspect ratio.

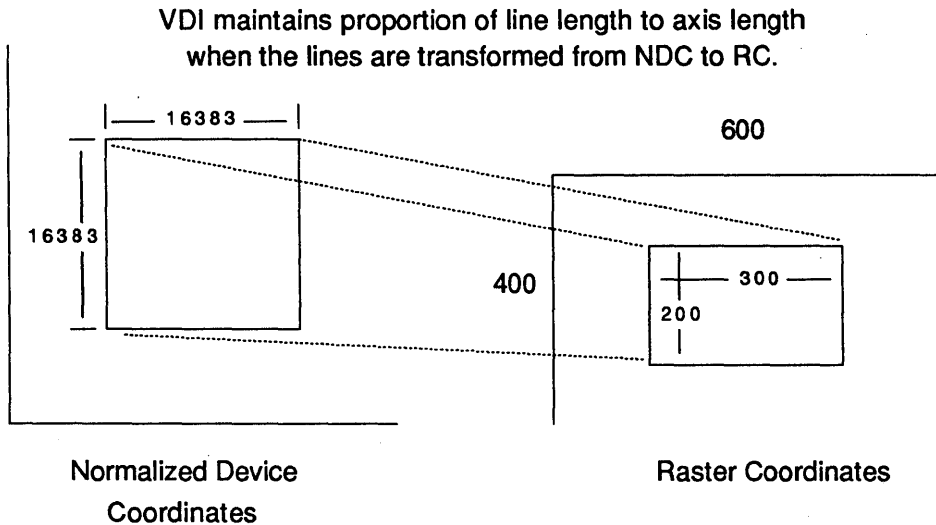


Figure 2-4. Aspect Ratio Conversion

2.5 Porting Applications to Different Environments

Programs written to GEM system software are portable to any machine with the same processor that can run GEM. Because the interface to the AES and VDI is machine-independent, you can also develop programs portable to GEM environments with different processor architectures. Portability considerations arise from three areas:

- memory management
- operating system file system functions
- processor word order

There are two memory management considerations:

- establishing application's runtime memory requirements
- allocating and deallocating temporary memory

To establish the application's runtime memory requirements, link its object file with one of several routines provided in the GEM Developer Kit. These routines determine the application's memory requirements and shrink the memory allocated at runtime to that amount.

You use the operating system's memory allocating and deallocating functions to get and release temporary memory. Note that GEM does not automatically release memory acquired through operating system calls.

One way to minimize machine-dependency is to put the environment-specific functions in include files. The PORTAB.H and MACHINE.H files provided in the GEM Developer Kit demonstrate include files you can develop to minimize program modifications.

2.6 Reserved Files

Table 2-1 lists the files, filenames, and file extensions reserved for GEM use. The metafile and image files mentioned in the table are defined as follows:

A **metafile** is a stored generic form of a picture file. The file is composed of a header which describes the file characteristics, such as coordinate system and physical page size, and a metafile item for each VDI function call made. Each item contains the function operation code (opcode), the number of input vertices in the function, an integer parameter count, a sub-opcode (if present), and the vertex and integer data.

An **image file** is a stored picture file in raw pixel form. The file contains a header and a series of scan line items. The header defines the source device characteristics including the pixel height and width, scan line width (number of pixels on the x axis), and the number of color planes. Each scan line item defines the pixel state for one or more rows of pixels.

See the VDI Reference Guide for the complete description of metafiles and image files and their use.

Table 2-1. Reserved GEM Files

Note: "afn" below means "any filename".

File Name	Description
afn.ACC	Desk accessory file
afn.APP	GEM application file
afn.DFN	Object definition file generated by GEM RCS (required only for editing RSC files)
afn.FNT	Font file
afn.GEM	Metafile (see below for description)
afn.IMG	Image file (see below for description)
afn.OUT	Merged text and graphics file
afn.RSC	Object tree resource file generated by GEM RCS
afn.RSH	Editable version of an RSC file
afn.SYS	Device driver file
ASSIGN.SYS	ASCII file used by GEM to get the <ul style="list-style-type: none">● device driver filename for a given device ID● filename for the device's font file See the <u>VDI Reference Guide</u> for the ASSIGN.SYS format description.
SCRAP.*	Temporary files created by the Scrap Library functions. See the Scrap function descriptions in the <u>AES Reference Guide</u> for the list of SCRAP-file extensions.

2.7 Enabling Graphics

You must load the GEM software before you can run programs that make AES and VDI calls. Once GEM is installed, there are three ways to load GEM and run an application:

- **Enable graphics and invoke GEM Desktop.** Use the following command to load GEM and invoke the GEM Desktop as the operating system shell:

GEM

This command makes GEM Desktop the default application. The user can invoke any application from the GEM Desktop. When the application terminates, the user returns to the GEM Desktop. When the GEM Desktop terminates, the user returns to the operating system shell.

- **Enable graphics and invoke a GEM application.** Use the following command to load GEM and invoke an application other than the GEM Desktop:

GEM filename

When the application terminates, the user returns to the operating system shell.

- **Enable graphics and invoke a non-GEM application.** Use the following command to load GEM and a non-GEM application, such as a debugger or test program, or a graphics program that uses VDI functions alone:

GEM /filename

For options 2 and 3, the file specified must be in the current search path or be preceded by the path specification.

End of Section 2

Application Programming Tasks and Examples

This section describes common programming tasks performed in GEM applications and demonstrates the use of AES and VDI functions. The examples are in C and taken from the DEMO.C program provided in the GEM Developer Kit. Before proceeding with the examples, run the executable version of this program, DEMO.APP, to see how it works. A summary of DEMO operation is provided below.

Note: DEMO is updated from time to time to incorporate new functions and correct bugs. Consequently, the version you receive might not be identical to the one used below.

The following tasks are described in this section:

- Program initialization
- Event management
- Menu processing
- Form processing
- Work area maintenance
- Program termination

A description of the differences between applications and accessories follows the program termination examples.

Note: In the descriptions and examples that follow, functions and variables are printed in lowercase letters and constants are printed in uppercase letters.

3.1 Application Planning

Before you begin coding, make the application's object tree resource file or files. This helps in planning and can save hours in development time through the use of the include files provided by GEM RCS.

To make a resource file, create a story board of the program's drop-down menus, panels, dialog boxes, alert boxes, and so forth. Next, use GEM RCS to construct each tree. You should endeavor to have all of

the program's object trees in a single resource file. As you create the trees, give a name to all objects you will reference in the application.

GEM RCS creates a number of files. It records the object trees in a file with the RSC extension. This is the file you load during program initialization. In addition, GEM RCS creates an include file for C, Pascal, BASIC, and/or FORTRAN-77 compilers with the index of each object you named. Finally, GEM RCS creates an object definition file with the extension DFN. You need this file only for editing the RSC file; you do not need it when you load the RSC file in the program.

3.2 DEMO Overview

The DEMO application is a program that draws or erases according to the mouse location and button state. It has four object trees:

- the menu bar with File, Options, and DEMO drop-down menus
- the Pen/Eraser Selection dialog box
- the DEMO information display
- the dialog box for the "Save as ..." File menu item

The names used in DEMO.C to reference the trees and individual objects are taken from the DEMO.H file provided in the GEM Developer Kit.

The File menu options are as follows:

- load a previously recorded file
- save the current screen in the open file
- save the current screen under a different filename
- abandon the changes made since last save
- quit DEMO

The Options menu provides items that select the Pen/Eraser Selection dialog and clear the screen

DEMO displays a cross-hair mouse form that echos mouse input. When the user holds down the leftmost mouse button, either a line is drawn or a square block is erased. The user selects the pen size and color or the eraser block size from the dialog box displayed when he or she clicks on the Pen/Eraser Selection item in the Options menu.

3.3 Program Initialization

The initialization tasks for a GEM application are as follows. The functions used to perform them are shown in parenthesis.

- Establish internal data structures in the AES (`appl_init`).
- Load the program's resource file (`rsrc_load`).
- Open screen virtual workstation to get handle and display device characteristics (`v_opnvwk` and `vq_extend`).
- Open output devices (`v_opnwk`) and load fonts (`vst_load_fonts`).
- Allocate temporary memory and declare the message buffer array.
- Initialize Memory Form Definition Blocks (MFDBs) and rectangle structures.
- Transform icons and bit image objects from the standard format to the device-specific format (`vr_trnfm`).
- Create and open the application's window (`wind_create` and `wind_open`).
- Display the menu bar (`menu_bar`), display title bar (`wind_set`), and sliders (`wind_set`).

During initialization, take control of the screen from the AES with the `wind_update` function and change the mouse form to an hourglass with `graf_mouse`. Generally, take control of the screen any time you update it to prevent the AES from updating it simultaneously. When initialization is complete, release control of the screen with another `wind_update` call and set the mouse form to the application's form with another `graf_mouse` call.

Listing 3-1 contains the DEMO.C code used to initialize the program and open the screen virtual workstation. The `BEG_UPDATE` argument in the `wind_update` call takes screen control from the AES and the `HOUR_GLASS` argument sets the mouse form to the hourglass form. The `wind_update` and `graf_mouse` calls that release control of the screen and change the mouse form to the program form are not shown in this listing.

Listing 3-1. Opening Virtual Workstation

```

WORD
demo_init()
{
    WORD    work_in[11];
    WORD    i;

    gl_apid = appl_init();           /* initialize libraries */
    if (gl_apid == -1)
        return(4);
    wind_update(BEG_UPDATE);
    graf_mouse(HOUR_GLASS, 0x0L);
    if (!rsrc_load( ADDR("DEMO.RSC") )) /* DEMO.RSC must be in path */
    {
        graf_mouse(ARROW, 0x0L);
        form_alert(1,
            ADDR("[3][Fatal Error !|DEMO.RSC|File Not Found][ Abort ]"));
        return(1);
    }
    /* open virtual workstation */
    for (i=0; i<10; i++)
    {
        work_in[i]=1;
    }
    work_in[10]=2;

    gem_handle = graf_handle(&gl_wchar,&gl_hchar,&gl_wbox,&gl_hbox);
    vdi_handle = gem_handle;
    v_opnvwk(work_in,&vdi_handle,work_out);

    if (vdi_handle == 0)
        return(1);
}

```

In the above listing, the `graf_handle` call provides the screen's handle required for the `v_opnvwk` call. When `v_opnvwk` returns, the `vdi_handle` value is redefined. The remainder of `DEMO` uses the new value for all VDI calls. The `graf_handle` call also returns the screen device's character cell and box dimensions. The box is a square large enough to hold the default system font character.

The `v_opnvwk` input arguments in the sample listing select the following:

- a virtual screen
- solid line
- dot polymarker
- system type face
- solid interior style
- black as the line, polymarker, text, and interior style color
- raster coordinates

DEMO opens no graphics devices other than the display device. If you need to load other device drivers and fonts, the time to do it is within the first ten AES calls and before an Event Library call. This guarantees that the memory allocated to the drivers and fonts is contiguous with the program memory. Similarly, make your operating system calls to get temporary memory before the tenth AES call or an event call to guarantee contiguous memory locations.

The `v_opnvwk` call returns the dimensions of the desktop window work area (`scrn_width` and `scrn_height`) in the `work_out` array. DEMO uses these values to initialize Memory Form Definition Blocks (MFDBs) for the world coordinate space (`undo_mfdb`) and the screen area (`scrn_mfdb`). The MFDB structure includes a pointer to the memory block, the form's x and y axis lengths, and the number of planes. (Planes are memory blocks the same size as the block defined in the MFDB. Multiple planes are necessary only for color screens.) The memory block pointer is always zero for the screen area. See the VDI Reference Guide for the description of MFDBs.

In DEMO, the world coordinate space and screen area are the same size, however, the view area is slightly smaller because of the application window's control areas and title bar.

Before you can display icons and bit image objects, you must transform them from standard form to the device-specific form. Listings 3-2 and 3-3 show the DEMO routines used for this purpose. There are six program-defined objects in DEMO: the three pen and three eraser sizes in the Pen/Eraser Selection dialog box.

Listing 3-2 shows getting the address of the object trees in which the pen and eraser sizes are a part (DEMOINF and DEMOPEND, respectively). The `R_TREE` value used in the `rscr_gaddr` calls is defined in `GEMBIND.H` as 0--the object type value for a tree. The first call to `trans_gimage` transforms the logo in the "About GEM Demo ..." dialog

box. In the loop that follows, `trans_gimage` transforms the three pen and three eraser images.

Listing 3-2. Transforming Objects and Initializing APPLBLK Structures

```

VOID
pict_init()
{
    LONG    tree;
    WORD    tr_obj, nobj;

    rsrc_gaddr(R_TREE, DEMOINFD, &tree);
    trans_gimage(tree, DEMOIMG); /* Xform logo in DEMO Info */
    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]));
    }
}

```

The `trans_gimage` function gets an icon's mask and data image addresses, height, and width from a data structure called a `ICONBLK`. Alternatively, it gets a bit image's image address, height, and width from a `BITBLK` data structure. (See the [AES Reference Guide](#) for the `ICONBLK` and `BITBLK` descriptions.) Listing 3-3 shows the transformation of the pen and eraser bit images.

Listing 3-3. Object Transformation

```

VOID
vdi_fix(pfd, theaddr, wb, h)      /* set up MFDB for xform */
    MFDB      *pfd;
    LONG      theaddr;
    WORD      wb, h;
{
    pfd->fww = wb >> 1;      /* # of bytes to words */
    pfd->fwp = wb << 3;      /* # of bytes to to pixels */
    pfd->fh = h;              /* height in scan lines */
    pfd->np = 1;              /* number of planes */
    pfd->mp = theaddr;        /* memory pointer */
}

WORD
vdi_trans(saddr, swb, daddr, dwb, h) /* 'on the fly' transform */
    LONG      saddr;
    WORD      swb;
    LONG      daddr;
    WORD      dwb;
    WORD      h;
{
    MFDB      src, dst;      /* local MFDB */

    vdi_fix(&src, saddr, swb, h);
    src.ff = TRUE;          /* standard format */

    vdi_fix(&dst, daddr, dwb, h);
    dst.ff = FALSE;        /* transform to device */
    /**/                    /* specific format */
    vr_trnfm(vdi_handle, &src, &dst);
}

VOID
trans_gimage(tree, obj)        /* xform bit images, icons */
    LONG      tree;
    WORD      obj;
{
    LONG      taddr, obspec;
    WORD      wb, hl, type;

    obspec = LLGET(OB_SPEC(obj));
    type = LWGET(OB_TYPE(obj));
}

```


Listing 3-3 (continued)

```

if ( type == G_ICON )
{
    taddr = LLGET(IB_PMASK(obspec)); /* pointer to icon mask*/
    wb = LWGET(IB_WB(obspec));
    wb = wb >> 3;                    /* pixels to bytes */
    hl = LWGET(IB_HL(obspec));      /* height in scan lines */
    vdi_trans(taddr, wb, taddr, wb, hl); /* transform mask */

    taddr = LLGET(IB_PDATA(obspec)); /* pointer to icon data*/
}
else
{
    taddr = LLGET(BI_PDATA(obspec)); /* pointer to image */
    wb = LWGET(BI_WB(obspec));      /* width in bytes */
    hl = LWGET(BI_HL(obspec));      /* height in scan lines*/
}
vdi_trans(taddr, wb, taddr, wb, hl); /* transform image or */
/**/                               /* icon data */
}

```

The next excerpt gets the address of the event message buffer, displays the application's menu bar, and creates the window. Window Library functions are used as follows to create the window:

- **wind_get**--Returns the location and size of the desktop window work area.
- **wind_create**--Creates a window to fill the desktop window work area and selects the title bar and all control areas (the information line is omitted).
- **wind_set**--Sets the title bar text string.

Listing 3-4. Set-up Message Buffer and Screen

```
ad_rmsg = ADDR((BYTE *) &gl_rmsg[0]); /* Get address of msg buf */
wind_get(DESK, WF_WXYWH, &gl_xfull, &gl_yfull, &gl_wfull, &gl_hfull);
rsrc_gaddr(R_TREE, DEMOMENU, &gl_menu); /* get menu address */
menu_bar(gl_menu, TRUE); /* show menu */
demo_whnd1 = wind_create(0x0fef, gl_xfull - 1, gl_yfull,
                        gl_wfull, gl_hfull);
if (demo_whnd1 == -1)
{
    form_alert(1, string_addr(DEMONWDW));
    return(3);
}
wind_set(demo_whnd1, WF_NAME, ADDR(wdw_title), 0, 0);
```

3.4 Event Management

After program initialization, use the Event Library calls to read user input and direct program execution. The event options are as follows:

- The user presses a key (evnt_keybd).
- The user presses one or more mouse buttons in a specific combination (evnt_button).
- The user moves the mouse into or out of a defined area (evnt_mouse).
- A message is waiting in the message buffer. The AES uses messages to indicate the following events:
 - a window control area was clicked on or dragged
 - a drop-down menu item was clicked on
 - a partial or complete screen redraw is necessary
 - a new window has been selected for display
- A specified number of ticks expires on the system clock (evnt_timer).

You specify a single event with the calls shown in parenthesis above. Use the `evnt_multi` function when you want to wait for one of several events to complete. You select the events and pass the related arguments in a single `evnt_multi` call.

The `evnt_multi` call returns when at least one of the events completes. The return code designates which events completed. To optimize message use, the AES merges events. Consequently, you can get a return code with more than one event flag set. Be sure to check for the completion of all events in your event servicing routine.

Listing 3-5 contains the DEMO's main event loop. DEMO has another event loop in the button handling routine; however, this is the primary loop for evaluating user input and determining how to proceed. The `evnt_multi` call in this listing specifies the following events:

- `MU_BUTTON` One to two clicks are made on the leftmost mouse button. A click is recorded when the button goes into the down state.
- `MU_MESAG` A message is received from the AES.
- `MU_M1` The mouse either enters or leaves, depending on the value of `m_out`, the application window work area.
- `MU_KEYBD` A key is pressed.

No timer event is specified in this call.

The `evnt_multi` call returns with an event type number indicating which event(s) occurred and the following information about the event:

- the mouse form's x and y coordinates (`mousex` and `mousey`)
- the state of the mouse buttons specified (`bstate`)
- the state of the keyboard's right and left shift, control, and, if present, ALT keys.

The proper event handler is selected by ANDing the `evnt_multi` return code with the event type values.

The `evnt_multi` call runs until one of the `hndl_` routines returns true. (`FOREVER` is defined in `PORTAB.H` as "for (;;)".) For example, this occurs when the close box (see DEMO's `hndl_msg` routine) or the Quit item is selected in the File menu (see DEMO's `hndl_menu` routine).

Listing 3-5. Main Event Loop

```

demo()
{
    BOOLEAN done;

    key_input = FALSE;
    done = FALSE;
    FOREVER      /* Infinite loop, defined as for(;;) */
    {
        ev_which = evnt_multi(
            MU_BUTTON | MU_MESAG | MU_M1 | MU_KEYBD,
            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, ad_rmsg, 0, 0,
            &mousex, &mousey, &bstate, &kstate,
            &kreturn, &bclicks);

        wind_update(BEG_UPDATE);

        if (!(ev_which & MU_KEYBD))
        {
            if (key_input)
            {
                curs_off();
                key_input = FALSE;
                save_work();
            }
        }

        if (ev_which & MU_MESAG)
        if (hdl_msg())
            break;
        if (ev_which & MU_BUTTON)
        if (hdl_button())
            break;
        if (ev_which & MU_M1)
        if (hdl_mouse())
            break;
        if (ev_which & MU_KEYBD)
        if (hdl_keyboard())
            break;

        wind_update(END_UPDATE);
    }
}

```

3.4.1 Button Handling

The `evnt_multi` call returns information in its output array that tells you which button or buttons were pressed and what their state was at the time. For all button events, the number of clicks is always at least one and never more than the number specified in the input array.

In DEMO's event loop, the button event specified is 1 or 2 clicks of the leftmost button. A click is defined as the transition of the button from up to down.

Listing 3-6 shows the button handling routine that draws lines or erases until the button is released. (In DEMO, erasing is actually outputting a rectangle in the background color.) The `x` and `y` arguments passed in `draw_pencil` are the mouse form's `x` and `y` coordinates when the button was pressed.

Another `evnt_multi` call is made within the button handler. The events stipulated and the program responses are as follows:

- Button is raised--stop drawing and return to the event loop.
- Timer expires--show mouse form and repeat this `evnt_multi` call.
- Mouse moves--draw line/eraser, update `x,y` array, and repeat `evnt_multi` call.

This listing also demonstrates use of the VDI attribute functions that define the writing mode; line width, end style, color, and type; and polygon (for the eraser) color and interior fill-style. The writing mode determines how the output affects the current display data. The options are replace, transparent, XOR, and reverse transparent. See the "Attribute Functions" description in the [VDI Reference Guide](#) for the explanation of the writing modes and the other attributes.

This listing also shows these other functions:

- `set_clip`: A local function that calls the `vs_clip` function to set and release the clipping rectangle.
- `save_work`: A local function that copies the current work area from the screen memory block to the `undo_mfdb` memory block.
- `eraser`: A routine that draws a rectangle at the current mouse form location.

Listing 3-6. Button Handling

```
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;

    set_clip(TRUE, &work_area);
    pxy[0] = x;
    pxy[1] = y;

    vs1_color(vdi_handle,demo_shade);
    vswr_mode(vdi_handle,MD_REPLACE);
    vs1_type (vdi_handle,FIS_SOLID);

    if (demo_shade != PEN_ERASER)
    {
        vs1_width (vdi_handle,demo_pen);
        vs1_ends(vdi_handle, 2, 2);
        hicount = 0;
        locount = 125;
        mflags = MU_BUTTON | MU_M1 | MU_TIMER;
        graf_mouse(M_OFF, 0x0L);
    }
    else
    {
        vsf_interior(vdi_handle, 1);
        vsf_color(vdi_handle, WHITE);
        mflags = MU_BUTTON | MU_M1;
    }
}
```

Listing 3-6 (continued)

```

done = FALSE;
while (!done)
{
    ev_which = evnt_multi(mflags, 0x01, 0x01, 0x00, 1,
        pxy[0], pxy[1], 1, 1, 0, 0, 0, 0, 0,
        ad_rmsg, locount, hicount, &pxy[2], &pxy[3],
        &bbutton, &kstate, &kreturn, &breturn);
    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];
        }
    } /* while */
    set_clip(FALSE, &work_area);
    save_work(); /* copy work area to undo */
}

```

3.4.2 Mouse Handling

A mouse event occurs when the mouse form enters or leaves a defined region. You can specify one region in the `evt_mouse` call or two regions in the `evt_multi` call. Both calls return the mouse form's `x` and `y` coordinates when the event completed.

One use of the mouse event is to change the mouse form as it enters and leaves the application window work area. The AES automatically converts the mouse form to an oblique arrow when the mouse leaves the application window. The application is responsible for all changes within the application window.

Note: When the user moves the mouse form into the menu bar and back onto the work area **without** touching a menu title, the AES treats the action as a mouse event. When a title is touched, however, the event is not complete until the user clicks the button. If the user clicks on a menu item, the AES returns a message event. If the user clicks on the application window work area, the AES returns a mouse event.

The `evt_multi` call in Listing 3-5 specifies one region: the application window work area. Listing 3-7 shows the routine that switches the mouse form back and forth between the application form and the oblique arrow as the mouse form goes between work area and window control areas.

Listing 3-7. Mouse Handling

```
WORD
hdl_mouse()
{
    BOOLEAN done;

    if (m_out)
        graf_mouse(ARROW, 0x0L);
    else
        graf_mouse(monumber, mofaddr);

    m_out = !m_out;
    done = FALSE;
    return(done);
}
```


3.4.3 Message Handling

The AES sends a message for the following events:

- a window control area was clicked on
- a menu item was clicked on
- a portion of the screen needs to be redrawn
- the user selected a new top window

A message is a 16-word array in which word(0) contains a number indicating message type, word(1) contains the application identifier (the `apl_init` return code) of the application that sent the message and word(2) indicates the length of the message. The `ap_id` of the AES is zero.

Note: The word(2) value is zero for the standard length, 16-word message. All messages from the AES are 16-words. Longer messages can be used for interapplication communication. See the Application Library description in the [AES Reference Guide](#) for the explanation of interapplication messages.

The remainder of the message array is message-type dependent. For example, the message indicating the selection of a menu item contains the object index of the menu title and the item selected. The redraw message, on the other hand, defines the area to redraw.

Listing 3-8 shows the portion of the DEMO message handler that interprets the following messages:

- `MN_SELECTED`: A menu item has been selected.
- `WM_REDRAW`: A portion of the screen needs to be redrawn.
- `WM_TOPPED`: A new window has been moved to the top.
- `WM_CLOSED`: The close box was clicked on.
- `WM_FULLED`: The full box was clicked on.

The portion of the listing that shows arrow box, slide bar, size box, and move message handling is described in Section 3.7 below.

The message array in `hdl_msg`, `gl_rmsg[8]`, is declared in DEMO's Local Data Structures. The pointer to it is defined in `demo_init` (see Listing 3-4) and passed to the AES in the `evnt_multi` call.

The `do_redraw` and `do_full` functions are DEMO routines that redraw the screen and toggle between full and partial screen views. The area

to be redrawn is taken from the message and passed to `do_redraw` in the input arguments. The `do_full` routine, on the other hand, must determine whether to display the full or partial view area. The `do_redraw` routine is shown in Listing 3-16 on page 3-34 and `do_full` appears in Listing 3-18 on page 3-37.

The `wind_set` function called in response to the `WM_TOPPED` message is an AES function that displays and makes active the window specified by the handle in word three of the message array.

Listing 3-8. Message Handling

```
BOOLEAN hndl_msg()  
{  
    BOOLEAN done;  
    WORD    wdw_hndl;  
    GRECT   work;  
  
    done = FALSE;  
    wdw_hndl = g1_rmsg[3];  
    switch( g1_rmsg[0] )  
    {  
    case MN_SELECTED:  
        done = hndl_menu(wdw_hndl, g1_rmsg[4]);  
        break;  
    case WM_REDRAW:  
        do_redraw(wdw_hndl, (GRECT *) &g1_rmsg[4]);  
        break;  
    case WM_TOPPED:  
        wind_set(wdw_hndl, WF_TOP, 0, 0, 0, 0);  
        break;  
    case WM_CLOSED:  
        done = TRUE;  
        break;  
    case WM_FULLED:  
        do_full(wdw_hndl);  
        break;  
        .  
        .  
        .  
    return(done)  
}
```

Note that the response to WM_CLOSED is to set the done value to TRUE. This causes a break in the event handler that results in the processing of DEMO'S termination routine. Also note that the use of the variable name wdw_hndl in the MN_SELECTED response is a misnomer in this context; gl_rmsg[3] in a MN_SELECTED message is the object index of the menu title selected.

3.4.4 Keyboard Handling and Text Output

The keyboard handling routine gets the character input from word(5) of evnt_multi's output array. Unless you intend otherwise, the keyboard handling routine should trap the system's program termination key and any other special or reserved keys; neither the AES nor VDI filter input for special characters.

Listing 3-9 illustrates how DEMO filters for CTRL-C (03H) and sets text attributes and output location. If the character is a CTRL-C, the routine returns TRUE. Like the WM_TOPPED case, this causes a break in the event handler and results in DEMO termination routine processing.

The text output attributes set in this routine are as follows:

- vswr_mode: replace writing mode
- vst_color: text color
- vst_height: character height
- vst_alignment: left justified along the bottom of the character cell

The vst_color and vst_height are selected by the user in the Pen/Eraser Selection dialog. The color argument is by color index. The character height argument is by raster units. In this case, the height value is dependent on the size of the pen or eraser selected.

The vst_height function returns the character and character cell dimensions supported by the device that are closest to the character height requested. Similarly, the vst_alignment function returns the alignment supported. The character and cell dimensions are used to increment the character and line spacing as each character and row is output. The data returned from vst_alignment is ignored in DEMO.

The graf_mkstate call returns the current mouse form location and the button and keyboard status information. Only the mouse form x,y data is used in DEMO.

Listing 3-9. Checking for CTRL-C and Setting Text Attributes

```

WORD
hnd1_keyboard()
{
    WORD    i;
    BYTE    str[2];
    GRECT   ltr, test;

    if ((str[0] = kreturn) == 0x03)
        return(TRUE);
    graf_mouse(M_OFF, 0x0L);
    if (!key_input)          /* Set FALSE in demo() */
    {
        vswr_mode(vdi_handle, MD_REPLACE);
        vst_color(vdi_handle, pen_shade);
        vst_height(vdi_handle, demo_height,
                  &gl_wchar, &gl_hchar, &gl_wbox, &gl_hbox);
        gl_hspace = gl_hbox - gl_hchar;
        vst_alignment(vdi_handle, 0, 3, &i, &i);
        graf_mkstate(&key_xbeg, &key_ybeg, &i, &i);
        key_xcurr = ++key_xbeg;
        key_ycurr = --key_ybeg;
    }
    else
        curs_off();
    str[1] = '\0';
}

```

DEMO uses the `vst_height` function to set the character height. Use this function to specify the character height in NDC or RC units. Alternatively, you can use the `vst_point` function to set the character height using points. (There are 72 points in an inch.) Both functions return the closest cell and character heights, in NDC or RC units, supported by the driver; `vst_height` also returns the cell height selected in points. The `intout(5)` value in the `v_opnwk` and `v_opnvwk` output array indicates how many character heights the device supports.

The last statement above sets the last character in the output string to NUL to satisfy the `v_gtext` function's character output requirements. Listing 3-10 shows the character output call sequence.

Listing 3-10. Character Output

```
set_clip(TRUE, &work_area);
v_gtext(vdi_handle, key_xcurr,
        key_ycurr, str);
set_clip(FALSE, &work_area);
key_xcurr += gl_wbox;
```

The last statement above increments the character cursor (called the "soft cursor" in DEMO) location by one character cell so that the next character received is output to the proper location.

3.5 Menu Processing

The `evnt_multi` call returns the menu title's and item's object index in a message when the user selects an item. These values indicate the offsets within the menu object tree of the title's and the item's OBJECT structure. (Section 3.6 below describes the OBJECT structure.)

Listing 3-11 shows the DEMO menu handling routine. In this excerpt, the names used in the case functions are the title and item names defined in DEMO.H; the DEMO include file generated by GEM RCS when the object trees were constructed. The `hndl_menu` title and item arguments are the corresponding index values passed in from the message handler (see Listing 3-8).

The `menu_tnormal` call is made here to reset the menu title to normal video. Although the AES redraws the screen where the drop-down menu was displayed, it does not reset the menu title.

Listing 3-11. Menu Handling

```
WORD
hdl_menu(title, item)
WORD title, item;
{
    WORD done;

    graf_mouse(ARROW, 0x0L);
    done = FALSE;
    switch (title) {
    case DEMODESK:
        if (item == DEMOINFO)
            do_about();
        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:
            done = TRUE;
            break;
        }
    }
}
```

Listing 3-11 (continued)

```
case DEMOOPTS:
    switch (item)
    {
        case DEMOPENS:
            do_penselect();
            break;
        case DEMOERAP:
            do_erase();
            break;
    }
}
menu_tnormal(gl_menu,title,TRUE);
graf_mouse(monumber, mofaddr);
return (done);
```

3.6 Form Processing

There are three phases to form processing:

1. the display and removal of the form's dialog box or panel
2. the processing of user interaction with the form
3. the retrieval of the data from the object structures

Display and Removal: You display the form with a sequence of `form_dial` and `objc_draw` calls. You call `form_dial` to tell the AES where the dialog box or panel will be displayed and `objc_draw` to draw the object tree. Optionally, you can make a `form_center` call before the `form_dial` call to get the x and y coordinates that will center the object tree on-screen.

The `form_dial`, `form_center`, and `objc_draw` calls require you to specify the location of the upper lefthand corner, height, and width of the form's root box. This information is available from the root object's OBJECT structure. You get the address of this structure with the `rsrc_gaddr` function.

Figure 3-1 illustrates the OBJECT structure. The fields are defined as follows. In all cases, nil is defined as -1.

- **ob_next**--Index of object's next sibling or, if no sibling, the parent. The root object has nil in this field.
- **ob_head**--Index of object's first child. Field is nil for objects with no children.
- **ob_tail**--Index of object's last child. Field is nil for objects with no children.
- **ob_type**--type of object
- **ob_flags**--bit map of object flags
- **ob_state**--bit map of object states
- **ob_spec**--object dependent information
- **ob_x**--x coordinate of object relative to parent*
- **ob_y**--y coordinate of object relative to parent*
- **ob_width**--width, in pixels, of the object
- **ob_height**--height, in pixels, of the object

* Location of object's upper lefthand corner

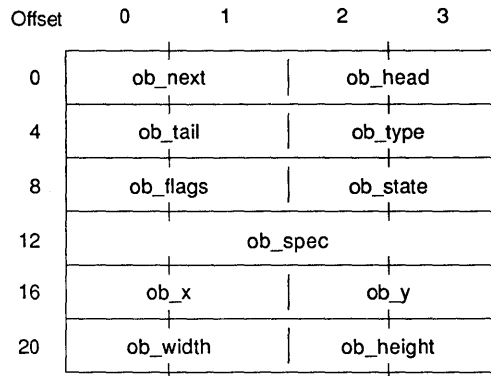


Figure 3-1. OBJECT Structure

You also use the `form_dial` function to end the display of the form. As in the other Form Library calls, you provide the coordinates and dimensions of the area to be released. The AES uses this information in the `WM_REDRAW` message it sends back to the application with the next `evnt_mesag` or `evnt_multi`.

Input Processing: You have two input processing options:

- You can use the `form_do` function to have the AES monitor user interaction and return when the user selects an exit object. This function does not allow you to interpret input data as it is input nor modify the object state while the form is processed.
- You can use the `form_keybd` and `form_button` functions to monitor user interaction as he or she moves from object to object in the object tree. These functions allow you to interpret data as it is input and modify the object state during form processing.

The `form_do` function takes the tree address and a starting object index and returns the index of the exit object selected. The AES registers user input in the OBJECT structure of the objects affected.

The `form_keybd` function takes the tree address, a character, and a starting object index. It returns zero if the character was a RETURN (ODH) or one if the character was anything but a RETURN. The `form_keybd` output array contains the next object selected and, if the input character was not a TAB, BACKTAB, UP, DOWN, or RETURN, the character input. (The significance of these characters is described in the Form Library section of the [AES Reference Guide](#).)

The `form_button` function takes the tree address, a starting object's index, and a number of clicks as input and returns when the number of clicks specified is entered. The output array contains a code and the index of the object selected. The code is one if the object selected is not an exit button or zero if the object is an exit button.

Retrieving Data: Retrieving the data after a `form_do` is a matter of comparing the original OBJECT structure contents with the new contents. For `form_keybd` and `form_button`, data retrieval is immediate rather than after the whole form is processed.

After getting the data, be sure to reset any fields that must be reset. For example, you must reset the SELECTED state flag on the exit button selected so that the next time the form is displayed the button is appears in normal rather than reverse video.

The following examples illustrate displaying, processing, and retrieving the data from DEMO's Pen/Eraser Selection dialog box. Figure 3-2 illustrates the dialog box with the medium pen size selected.

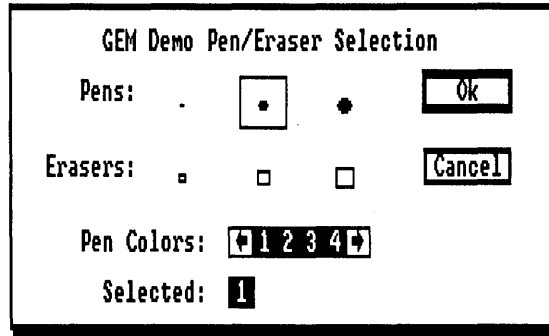


Figure 3-2. DEMO Pen/Eraser Selection Dialog Box

Listing 3-12 shows how to get an object tree's address and set the current selections. DEMOPEND is the index for this object tree's root object as defined in DEMO.H. The PEN_FINE, PEN_MEDIUM, and PEN_BROAD represent the three size options.

The rsrc_gaddr function returns the address of the root object in the &tree argument. The switch function and sel_obj routine determine which pen or eraser size is currently defined by demo_pen and set the SELECTED state flag in the corresponding OBJECT structure. The set_color call sets the current color selected.

Listing 3-12. Setting the Dialog Box's Current Selections

```

VOID
do_obj(tree, which, bit) /* set specified bit in object state */
LONG   tree;
WORD   which, bit;
{
    WORD   state;

    state = LWGET(OB_STATE(which));
    LWSET(OB_STATE(which), state | bit);
}

VOID
sel_obj(tree, which) /* turn on selected bit of spcfd object */
LONG   tree;
WORD   which;
{
    do_obj(tree, which, SELECTED);
}

rsrc_gaddr(R_TREE, DEMOPEND, &tree);
/**/ /* first setup 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;
}
set_color(tree, DEMOPCLR, pen_shade, bind);

```

Listing 3-13 demonstrates the display of a dialog box. The `form_center` command takes the root object's address and returns the `x` and `y` coordinates of its upper lefthand corner and the box's width and height. Use the tree's corner coordinates and dimension values to reserve the space in the first `form_dial` call, to set the clip rectangle in

`objc_draw`, and finally to release the space in the last `form_dial` call. The `exitobj` value returned by `form_do` is the object index of the exit button selected.

Note: The `form_dial` function takes two sets of rectangle coordinates and dimensions. Put the arguments for the rectangle to be reserved and released in the second set.

The AES takes the coordinates and dimensions passed in the last `form_dial` call shown in Listing 3-13 and issues a `WM_REDRAW` message based on these values. You use this information to replace the dialog box with the former screen contents.

Listing 3-13. Display and Processing of a Dialog Box

```
WORD
hndl_dial(tree, def, x, y, w, h)
LONG   tree;
WORD   x, y, w, h, def;

    def = 0;
{
    WORD   xdial, ydial, wdial, hdial, exitobj;
    WORD   xtype;

    form_center(tree, &xdial, &ydial, &wdial, &hdial);
    form_dial(0, 0, 0, 0, 0, xdial, ydial, wdial, hdial);
    objc_draw(tree, ROOT, MAX_DEPTH, xdial, ydial, wdial, hdial);

    FOREVER
    {
        exitobj = form_do(tree, def) & 0x7FFF;
        xtype = LWGET(OB_TYPE(exitobj)) & 0xFF00;
        if (!xtype)
            break;
        if (xtend_do(tree, exitobj, xtype))
            break;
    }

    form_dial(3, 0, 0, 0, 0, xdial, ydial, wdial, hdial);
    return (exitobj);
}
```

The routine in the FOREVER loop is an advanced bit of form processing using the high order byte of the object's `ob_type` value. The AES ignores this byte, however, you can use it. For example, here it is used to designate the pen color objects as a special type of exit button.

Listing 3-14 shows how DEMO determines whether the OK or Cancel exit button was selected and, if the former, gets the color and pen or eraser size selected. The DEMOPFIN and DEMOEBRD are the beginning and ending indexes for the pen and eraser objects in the tree. The LWGET function returns the word stored in the designated OBJECT structure field (in this case, the object state).

Listing 3-14. Getting Data and Resetting SELECTED Flag

```

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);
}

VOID
desel_obj(tree, which) /* turn off selected bit of spcfd object */
LONG    tree;
WORD    which;
{
    undo_obj(tree, which, SELECTED);
}

for (pse1_obj = DEMOPFIN; pse1_obj <= DEMOEBRD; pse1_obj++)
    if (LWGET(OB_STATE(pse1_obj)) & SELECTED)
    {
        desel_obj(tree, pse1_obj);
        break;
    }
color = get_color(tree, DEMOPCLR);

```

Listing 3-14 (continued)

```
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);
```

3.7 Work Area Maintenance

Work area maintenance refers to three tasks, all of them initiated by a message from the AES:

- redrawing a specified portion of the screen (WM_REDRAW)
- sizing and moving the window and switching between full and partial sizes (WM_SIZED, WM_MOVED, and WM_FULLED)
- relocating the view area (WM_ARROWED, WM_HSLID, and WM_VSLID)

The last two tasks are required only if you select the full and size boxes, move bar, sliders, and arrows when you create the window.

Common to all of these tasks is the use of a block of memory that contains an up-to-date copy of the image in the screen's memory block. In DEMO, the memory block is defined by an MFDB called `undo_mfdb` created during program initialization. DEMO updates the undo buffer from the screen block with every button and keyboard event.

The function used to copy between buffers is the VDI `vro_copyfm` (Copy Raster, Opaque) function. Listing 3-15 shows the `vro_copyfm` call and the preparatory routines. The sequence of events preceding the call is as follows:

1. Copy the current work area coordinates and dimensions into a temporary rectangle structure (`rc_copy`).
2. Determine the intersection of the screen area and the temporary rectangles and put the coordinates and dimensions of the overlap into the temporary rectangle structure (`rc_intersect`). The `min` and `max` routines determine which of the two arguments provided is smaller and larger, respectively.
3. Turn off the mouse form (`graf_mouse`).
4. Copy the rectangle defined in the temporary structure from the screen memory block to the `undo_mfdb` memory block (`rast_op`).
5. Turn on the mouse form (`graf_mouse`)

Listing 3-15. Save Work Area

```

VOID
save_work()      /* copy work_area to undo_area buffer  */
{
    GRECT    tmp_area;

    rc_copy(&work_area,&tmp_area);
    rc_intersect(&scrn_area,&tmp_area);
    graf_mouse(M_OFF, 0x0L);
    rast_op(3, &tmp_area, &scrn_mfdb, &undo_area, &undo_mfdb);
    graf_mouse(M_ON, 0x0L);
}

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;
}

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) );
}

```


Listing 3-15 (continued)

```

VOID          /* bit block level trns */
rast_op(mode, s_area, s_mfdb, d_area, d_mfdb)
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);
}

VOID
grect_to_array(area, array) /* convert x,y,w,h to upper left x,y */
GRECT   *area;             /* and 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;
}

```

Redrawing: The AES sends a WM_REDRAW to indicate that an area of the screen needs to be redrawn. The area to be redrawn is defined in the message array's word(4) through word(7) as follows:

- word(4): x screen coordinate of the upper lefthand corner
- word(5): y screen coordinate of the upper lefthand corner
- word(6): width in pixels of the rectangle
- word(7): height in pixels of the rectangle

Recall that the message also provides the window handle in word(3).

Listing 3-16 redraws the screen area defined in the message. The `do_redraw` function is called from the message handler (see Section 3.4.3 above) as follows:

```
do_redraw(wdw_hndl, (GRECT *)&gl_rmsg[4]);
```

The values for the window handle and rectangle location and size arguments are taken from the message itself.

An important task in redrawing the window is to determine what portion of the rectangle provided in the message intersects with each rectangle in the window's list of rectangles. A window gets divided into multiple rectangles as other windows and boxes (dialog, error, alert, and form) are displayed. Figure 3-3 illustrates the resulting rectangles when a window is overlaid in the middle of another window. The AES maintains the rectangle list.

The AES breaks up window A into four rectangles while window B is displayed in the middle.

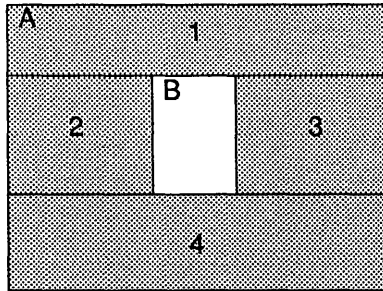


Figure 3-3. Window Rectangles

You get the coordinates and size of each rectangle in the list through repeated `wind_get` calls. Listing 3-16 shows the use of `wind_get` to get the first and all subsequent rectangles in the list. The AES returns a rectangle with a width and height of zero when there are no more rectangles on the list.

In DEMO, the redraw is performed by copying from the `undo_mfdb` memory block to the `scrn_mfdb` memory block. In Listing 3-16, the copy is done by the `rast_op` function. See Listing 3-15 above for the description of `rast_op` and the `rc_copy` and `rc_intersect` functions.

Listing 3-16. Redrawing a Portion of the Screen

```

VOID
do_redraw(wh, area)      /* redraw specified area from undo bfr */
WORD   wh;
GRECT  *area;
{
    GRECT  box;
    GRECT  dirty_source, dirty_dest;

    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 (wh == demo_whnd1)
            {
                rc_copy(&box, &dirty_dest);
                if (rc_intersect(&work_area, &dirty_dest))
                {
                    dirty_source.g_x = (dirty_dest.g_x - work_area.g_x)
                                        + undo_area.g_x;
                    dirty_source.g_y = (dirty_dest.g_y - work_area.g_y)
                                        + undo_area.g_y;
                    dirty_source.g_w = dirty_dest.g_w;
                    dirty_source.g_h = dirty_dest.g_h;
                    rast_op(3, &dirty_source, &undo_mfdb,
                            &dirty_dest, &scrn_mfdb);
                }
            }
        }
        wind_get(wh, WF_NEXTXYWH, &box.g_x, &box.g_y, &box.g_w, &box.g_h);
    }
    graf_mouse(M_ON, 0x0L);
}

```

The `graf_mouse` function is called first to turn off the mouse during the redraw and then to turn it back on. If you do not turn off the mouse and it is in the redraw area, it is overwritten. Moving the mouse subsequently leaves a hole on the screen.

Sizing, Moving, and Switching: The user can move a window (drag the title bar down and to the right), size it (drag the size box up and to the left), and switch between full and partial views. The AES reports these user actions to the application in the form of WM_SIZED, WM_MOVED, and WM_FULLED messages, respectively. The WM_SIZED and WM_MOVED messages provide the window handle, the requested location, and the requested height and width. The WM_FULLED message contains the window handle alone.

You change the window size and location with the `wind_set` function. Besides the size and location, you also use `wind_set` to change the title and information bar contents, the slider location and size, and the top window. For WM_SIZED and WM_MOVED, you use `wind_set` as follows:

- WM_SIZED: Set the size of the current window and the slide bar size and location. The window location remains the same.
- WM_MOVED: Set the location of the current window. The window size remains the same.

Listing 3-17 shows DEMO's WM_SIZED and WM_MOVED message responses. The `gl_rmsg` array contains the message contents. In the WM_SIZED case, the `wind_calc` function is used first to get the location and size of the application window's work area alone and a second time to get the entire window's size and location, with the window control areas included. The `align(x)` function forces word alignment for the column position. The `set_work` function orients the `undo_mfdb` view area to the current window work area. (see Listing 3-19 below for the `set_work` description).

Note: Use the `align` function or an equivalent when program performance is more important than exact window placement. The best performance results when you "snap" to word boundaries.

Listing 3-17. WM_SIZED and WM_MOVED Message Responses

```

case WM_SIZED:
    wind_calc(1, 0x0fef, g1_rmsg[4], g1_rmsg[5], g1_rmsg[6],
             g1_rmsg[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_calc(0, 0x0fef, work.g_x, work.g_y, work.g_w, work.g_h,
             &g1_rmsg[4], &g1_rmsg[5], &g1_rmsg[6], &g1_rmsg[7]);
    wind_set(wdw_hnd1, WF_CXYWH, g1_rmsg[4],
             g1_rmsg[5], g1_rmsg[6], g1_rmsg[7]);
    set_work(TRUE);
    break;
case WM_MOVED:
    g1_rmsg[4] = align_x(g1_rmsg[4]);
    wind_set(wdw_hnd1, WF_CXYWH, align_x(g1_rmsg[4]) - 1,
             g1_rmsg[5], g1_rmsg[6], g1_rmsg[7]);
    set_work(FALSE);
    break;

```

Listing 3-18 shows DEMO's response to a WM_FULLED message. The argument passed to this routine is the window handle provided in the message. As with WM_SIZED and WM_MOVED, you use wind_set to redimension the window. However, before you redimension a window, you must determine whether to display the full or the reduced size.

As shown in Listing 3-18, you use the wind_get function to determine the toggle status of the full box by comparing the coordinates of the following windows:

- WF_CXYWH: current window
- WF_PXYWH: previous window
- WF_FXYWH: full window

The current window location and size is set by each wind_set call that specifies the W_CXYWH input argument (see Listing 3-17). The AES automatically keeps track of the full and previous window location and size.

Listing 3-18. WM_FULLED Message Response

```

VOID
do_full(wh)      /* make window either full size or return */
WORD   wh;      /* to previous shrunken size                */
{
    GRECT   prev;
    GRECT   curr;
    GRECT   full;

    graf_mouse(M_OFF,0x0L);
    wind_get(wh, WF_CXYWH, &curr.g_x, &curr.g_y, &curr.g_w, &curr.g_h);
    wind_get(wh, WF_PXYWH, &prev.g_x, &prev.g_y, &prev.g_w, &prev.g_h);
    wind_get(wh, WF_FXYWH, &full.g_x, &full.g_y, &full.g_w, &full.g_h);
    if ( rc_equal(&curr, &full) )
    {
        /* is full now so change*/
        /**/                /* to previous */
        wind_set(wh, WF_CXYWH, prev.g_x, prev.g_y, prev.g_w, prev.g_h);
        rc_copy(&save_area, &undo_area);
        set_work(TRUE);
    }
    else
    {
        /* is not full so make */
        /**/                /* it full */
        rc_copy(&save_area, &undo_area);
        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);
}

```

Changing the View Area

The user clicks on an arrow box, clicks on the scroll bar, or drags the vertical or horizontal slider to change the view area. The AES sends a message to the application to indicate that a view area change has been requested. The information provided in the messages is as follows:

- **WM_ARROWED:** A number indicating the arrow box or scroll bar location clicked on as follows:

- 0 = page up (scroll bar above slider)
- 1 = page down (scroll bar below slider)
- 2 = row up (up arrow)
- 3 = row down (down arrow)
- 4 = page left (scroll bar left of slider)
- 5 = page right (scroll bar right of slider)
- 6 = column left (left arrow)
- 7 = column right (right arrow)

The amount of view area change equivalent to a page, row, or column selection is program defined.

- **WM_HSLID:** A number in the range 1 to 1000 indicating the new location of the horizontal slider's left edge where

- 1 = the leftmost position
- 1000 = the rightmost position

- **WM_VSLID:** A number in the range 1 to 1000 indicating the new location of the vertical slider's upper edge where

- 1 = the top position
- 1000 = the bottom position

Once you get the message, you have to interpret the data, reset the slider positions to reflect the portion of the window selected, and change the view area. You use the `wind_set` function to change slider location and size. To change the view area, DEMO calls `vro_cpyfm` to copy the new view area from the `undo_mfdb` memory block to the screen's memory block.

Listing 3-19 shows how DEMO interprets the data and changes the slider location and size and view area. Listing 3-20 shows the `set_work` and `restore_work` routines called in Listing 3-19.

Listing 3-19. Arrow and Slider Message Responses

```
case WM_ARROWED:      /* arrow or scroll bar clicked */
    switch(gl_rmsg[4])
    {
    case WA_UPPAGE:
        undo_area.g_y = max(undo_area.g_y - undo_area.g_h, 0);
        break;
    case WA_DNPAGE:
        undo_area.g_y += undo_area.g_h;
        break;
    case WA_UPLINE:
        undo_area.g_y = max(undo_area.g_y - YSCALE(16), 0);
        break;
    case WA_DNLINE:
        undo_area.g_y += YSCALE(16);
        break;
    case WA_LFPAGE:
        undo_area.g_x = max(undo_area.g_x-undo_area.g_w, 0);
        break;
    case WA_RTPAGE:
        undo_area.g_x += undo_area.g_w;
        break;
    case WA_LFLINE:
        undo_area.g_x = max(undo_area.g_x - 16, 0);
        break;
    case WA_RTLINE:
        undo_area.g_x += 16;
        break;
    }
    set_work(TRUE);
    restore_work();
    break;
case WM_HSLID:      /* horizontal slider dragged */
    undo_area.g_x = align_x(UMUL_DIV(undo_mfdb.fwp - undo_area.g_w,
    gl_rmsg[4], 1000));
    set_work(TRUE);
    restore_work();
    break;
case WM_VSLID:      /* vertical slider dragged */
    undo_area.g_y = UMUL_DIV(undo_mfdb.fh - undo_area.g_h,
    gl_rmsg[4],1000);
    set_work(TRUE);
    restore_work();
    break;
```


Listing 3-20. Updating the undo_mfdb, Sliders, and Screen

```

VOID
set_work(slider_update)      /* update undo area, clamping to page */
BOOLEAN slider_update;      /* edges, and updt sliders if req'd */
{
    WORD          i;

    wind_get(demo_whnd1, WF_WXYWH,
             &work_area.g_x, &work_area.g_y,
             &work_area.g_w, &work_area.g_h);

    undo_area.g_w = work_area.g_w;
    undo_area.g_h = work_area.g_h;
    /**/          /* clamp work area to page edges */
    undo_area.g_x = align_x(undo_area.g_x);
    if ((i = undo_mfdb.fwp - (undo_area.g_x + undo_area.g_w)) < 0)
        undo_area.g_x += i;
    if ((i = undo_mfdb.fh - (undo_area.g_y + undo_area.g_h)) < 0)
        undo_area.g_y += i;

    if (slider_update)
    {
        wind_set(demo_whnd1, WF_HSLIDE, UMUL_DIV(undo_area.g_x, 1000,
            undo_mfdb.fwp - undo_area.g_w), 0, 0, 0);
        wind_set(demo_whnd1, WF_VSLIDE, UMUL_DIV(undo_area.g_y, 1000,
            undo_mfdb.fh - undo_area.g_h), 0, 0, 0);
        wind_set(demo_whnd1, WF_HSLSIZ, UMUL_DIV(work_area.g_w, 1000,
            undo_mfdb.fwp), 0, 0, 0);
        wind_set(demo_whnd1, WF_VSLSIZ, UMUL_DIV(work_area.g_h, 1000,
            undo_mfdb.fh), 0, 0, 0);
    }

    /* only use portion of work_area on screen */
    rc_intersect(&scrn_area, &work_area);
    undo_area.g_w = work_area.g_w;
    undo_area.g_h = work_area.g_h;
}

```

Listing 3-20 (continued)

```
VOID
restore_work()          /* restore work_area from undo_area   */
{
    GRECT    tmp_area;

    rc_copy(&work_area,&tmp_area);
    rc_intersect(&scrn_area,&tmp_area);
    graf_mouse(M_OFF, 0x0L);
    rast_op(3, &undo_area, &undo_mfdb, &tmp_area, &scrn_mfdb);
    graf_mouse(M_ON, 0x0L);
}
```

3.8 Program Termination

Terminating a GEM application involves the following graphics-related tasks:

- Close and delete the application's window.
- Remove the menu bar.
- Close all devices opened.
- Release the internal data structures used by the application.

Besides the graphics-related tasks, you should perform the following operating system related tasks as required:

- Flush all output buffers.
- Release all memory allocated by the application.
- Close all open disk files.

Listing 3-21 shows the DEMO termination routine. The `term_type` argument is the `demo_init` return value. The `wind_update` function is used here to release control of the window because processing jumps to `demo_term` from a break in the main event handler--`demo()`. (Recall that the event loop begins with a `wind_update` to take control of the window.)

Listing 3-21. Program Termination Routine

```

demo_term(term_type)
WORD    term_type;
{
    switch (term_type)      /* NOTE: all cases fall through */
    {
        case (0 /* normal termination */):
            wind_close(demo_whnd1)
            wind_delete(demo_whnd1);
        case (3):
            menu_bar(0x0L, FALSE);
            dos_free(undo_mfdb.mp);
        case (2):
            v_clsvwk( vdi_handle );
        case (1):
            wind_update(END_UPDATE);
            appl_exit();
        case (4):
            break;
    }
}

```

3.9 Creating Accessories

You create accessories as you do applications. That is, you begin with GEM RCS to create the object trees and then build the program with AES and VDI functions.

For the most part, accessories and applications have the same program components and use the same AES and VDI functions. For example, in both application and accessory initialization, you use the following functions:

- `appl_init` to initialize the internal data structures
- `graf_handle` to get the VDI handle of the current screen workstation
- `v_opnvwk` to initialize the virtual workstation
- `wind_get` to find out the window 0 location and dimensions.

However, in a desk accessory, you also call `menu_register` to add the accessory's name to the accessory menu and you do not typically create and open the window. Creating and opening the window are

usually performed when the accessory receives the AC_OPEN message. Besides initialization, accessories differ in the following ways:

- **Event handling:** Accessories do not generally exit the event loop. Instead, they run "forever." The main event loop should be an `evnt_msg` call waiting for an AC_OPEN message. All other forms of input should be disabled to avoid conflict with the current application. Use the `evnt_multi` call to get mouse, button, and keyboard input only after the AC_OPEN has been received.
- **Message handling:** Accessories must support two messages not supported by applications:
 - AC_OPEN: Tells the accessory to display its window and start execution.
 - AC_CLOSE: Tells the accessory that the current application has closed, the screen is about to be cleared, and the window data structures will be cleared. (AC_CLOSE is not the accessory's terminate message.) Typically, you initialize the accessory's handle in response to this message. (The user might have closed the application before closing the accessory.)

Because the window data structures are cleared when an application terminates, you should create the accessory window with each AC_OPEN message. In addition, when you get a WM_CLOSE message, you should not terminate the accessory. Instead, close and delete the window and make the `evnt_msg` call, waiting for another AC_OPEN message.

Note: If an accessory is on-screen but covered completely by another window, the GEM Desktop manual instructs the user to select the accessory through the menu to get it back on-screen. This results in two successive AC_OPEN messages without an interceding WM_CLOSE message. You should interpret this AC_OPEN as the equivalent of a WM_TOPPED message and not create another accessory window.

- **Termination:** Accessories do not normally terminate. When the close box is clicked on, you close and delete the window, but there is no need to close the virtual workstation (`v_clsvwk`) or exit the application (`appl_exit`).

Note: If you do have an exit condition in an accessory, be sure to call the `menu_unregister` function in the termination routine to remove the name from the accessory drop-down menu.

- **Execution:** You cannot execute an accessory; therefore, you cannot debug it. To debug an accessory, you must compile and link the program as an application.

To accommodate the differences between applications and accessories, use conditional statements to select application-specific routines for debugging purposes and the accessory-specific routines when debugging is complete. The sample accessory HELLO.C demonstrates this technique.

HELLO.C displays a small window with the message, "Hello World." You control whether HELLO.C is compiled as an application or an accessory by defining the `DESKACC` value as 0 (application) or 1 (accessory). The value of `DESKACC` controls program flow in a series of if statements. For example, Listing 3-22 shows the if statement used in the HELLO event handler.

Listing 3-22. HELLO Event Handler

```
hello()
{
    BOOLEAN done;

    done = FALSE
    while( !done )
    {
        ev_which = evnt_mesag(ad_rmsg) /* wait for message */
        wind_update(BEG_UPDATE)      /* begin window update */
        done = hndl_mesag();          /* message type handler */
        wind_update(END_UPDATE)      /* end window update */
    }
    #if DESKACC
        done = FALSE; /* never exit loop for desk accessory */
    #endif
}
```

In this listing, when DESKACC is TRUE, HELLO is an accessory and the while loop never completes. When DESKACC is FALSE, HELLO is an application and the while loop is satisfied when done equals TRUE. HELLO message handling, termination, and initialization also demonstrate techniques for switching between application and accessory with DESKACC.

End of Section 3

Index

A

AC_CLOSE, 3-43

AC_OPEN, 3-43

ACC file, 2-10

Accessories, 2-4

 closing, 3-44

 menu_unregister, 3-44

AES, 1-2

 converting titles, 1-2

 creating desktop window, 1-2

 kernel, 2-2

 menu and alert buffer, 2-4

 mouse and button input, 1-2

 opening virtual workstation,
 1-2

 Screen Manager, 2-2

Alert box

 components, 1-10

Alert buffer, 2-4

Allocating memory

 when to, 3-5

APP file, 2-10

appl_exit, 3-41

appl_init, 3-3

Application Environment

 Services (see AES), 1-2

Application Library

 function summary, 1-13

Application space, 2-5

 driver allocation, 2-5

 font allocation, 2-5

 minimum, 2-5

 resource file allocation, 2-5

 temporary memory allocation,
 2-5

Application window, 1-4

 arrows, 1-5

 close box, 1-5

 components, 1-5

 control areas, 1-5

 creating, 1-4

 dialog box size, 1-8

 full box, 1-5

 full box switching, 3-36

 information components, 1-4

 information line, 1-4

 limits, 1-4

 move bar, 1-5

 overlapping, 1-4

 rectangle list, 3-33

 removing, 1-4

 scroll bar and slider, 1-5

 selecting control areas, 1-4

 selecting scroll bar and slider,
 1-7

 size box, 1-5

 size of, 1-3, 1-4

 title bar, 1-4

 when to release control over ,
 3-3

 when to take control over, 3-3

 work area, 1-7

 work area maintenance, 3-30

- work area size, 1-4
- work area size and location, 1-7
- Applications
 - before coding, 3-1
 - extension, 2-10
 - initialization tasks, 3-3
 - porting, 2-9
- Arrow box message, 3-38
- Arrows, 1-5
- Aspect ratio, 2-2, 2-8
 - affect on output, 2-8
 - compensating for, 2-8
- ASSIGN.SYS, 2-1, 2-10
 - device ID, 2-1
 - driver file extension, 2-10
 - font file extension, 2-10
- Attribute Functions, 1-19
 - fill style and color, 3-12
 - lines, 3-12
 - text, 3-18
 - use of, 3-12

B

- Bit image objects, 1-10
 - transformation, 3-5
- Button event
 - click definition, 3-12
 - number of clicks reported, 3-12
- Button handling, 3-12

C

- Character output, 3-19
- Click, 3-12
- Clipping, 3-12
- Close box, 1-5
- Coordinate systems, 2-7
 - selecting, 2-7
- Copy mode, 1-20
- Current window
 - setting coordinates, 3-36

D

- Desk accessories, 2-4
 - displaying name, 2-4
 - event handling, 3-43
 - extension, 2-10
 - initialization, 3-42
 - loading, 2-4
 - menu title, 2-4
 - message handling, 3-43
 - program execution, 3-44
 - program termination, 3-44
 - registering name, 3-42
- Desktop window
 - direct output to, 1-3
 - getting size, 3-8
 - handle, 1-3
 - height and width, 1-4
 - menu bar, 1-2
 - origin point, 1-4
 - work area, 1-3
- Device attributes, 1-18
- Device drivers, 2-1
 - in memory, 2-5

- selection, 2-1
- Device ID, 2-1
- DFN file, 2-10, 3-2
 - when needed, 3-2
- Dialog box
 - as form, 1-8
 - components, 1-10
 - modal, 1-8
 - modeless, 1-8
 - size, 1-8
 - types of, 1-8
- Dispatcher, 2-3
 - allocating CPU time, 2-3
 - dispatch events, 2-4
 - not-ready list, 2-4
 - ready list, 2-4
- Dispatching, 2-3
- Driver file extension, 2-10
- Driver space allocation, 2-5
- Drop-down menu
 - components, 1-10
 - redrawing screen, 1-2

E

- Escape functions, 1-20
- Event Library, 1-13
 - events supported, 1-13
 - function summary, 1-13
 - interprocess messages, 1-13
 - message system, 1-13
- evnt_button, 3-9
- evnt_keybd, 3-9
- evnt_mesag, 3-9
 - in HELLO, 3-44
- evnt_mouse, 3-9

- evnt_multi, 3-10
 - button handling loop, 3-12
 - in button handler, 3-14
 - main event loop, 3-10
 - output array, 3-10
 - return code, 3-10
- evnt_timer, 3-9
- Extended Graphics Library
 - function summary, 1-17

F

- File Selector Library
 - function summary, 1-16
- Files
 - reserved, 2-10
- FNT file, 2-10
- Font file extension, 2-10
- Fonts, 2-5
- Form Library
 - function summary, 1-15
- form_alert, 3-3, 3-8
- form_button, 3-24
- form_center, 3-27
- form_dial, 3-27
- form_do, 3-24
- form_keybd, 3-24
- Forms
 - automatic processing, 3-24
 - determining exit button, 3-28
 - display, 3-22
 - processing, 3-22
 - removal, 3-22
 - resetting selected objects, 3-24
 - retrieving data, 3-24

Full box, 1-5
Function Libraries, 1-13

G

GDOS, 2-5
GEM file, 2-10
GEM RCS, 1-11
 definition file extension, 2-10
 editable resource file, 2-10
 files produced, 3-2
 include file, 3-2
 resource file extension, 2-10
graf_handle, 3-3
 getting screen handle, 3-4
graf_mkstate, 3-18
graf_mouse
 during initialization, 3-3
Graphics Device Operating
 System, 2-5
Graphics Library
 function summary, 1-15

H

Handle, 1-18

I

Icons
 transformation, 3-5
Image files, 2-10
IMG file, 2-10
Information line, 1-4
 contents, 1-4

length, 1-4
Input Functions, 1-20
 modes, 1-20
Inquire functions, 1-20
Items (in menus), 1-2

K

Kernel, 2-2
Keyboard event, 3-18

L

Library, 1-2
Line attributes, 3-12
Loading fonts, 2-5

M

Memory Form Definition Block
 (see MFDB), 3-5
Menu and alert buffer, 2-4
 size, 2-4
Menu bar, 1-2
 components, 1-10
 controlling contents, 1-3
 drop-down menu, 1-2
 items, 1-2
 reporting selection, 3-20
 resetting title, 3-20
 titles, 1-2
Menu Library, 1-3
 function summary, 1-13
Menu processing, 3-20
menu_bar, 3-8

- clearing current tree, 3-41
- menu_normal, 3-20
- menu_unregister, 3-44, 3-42
- Message array, 3-16
- Message system, 1-13

Messages

- application to itself, 1-13
- between application and accessory, 1-13
- events, 3-9
- format, 3-16
- interapplication, 1-13
- MN_SELECTED, 3-16
- WM_CLOSED, 3-16
- WM_FULLED, 3-16
- WM_REDRAW, 3-16
- WM_TOPPED, 3-16

Metafile, 2-10

- extension, 2-10

MFDB

- initialization, 3-5

MN_SELECTED, 3-16

Mouse driver, 2-5

Mouse event, 3-15

- changing mouse form, 3-15
- from menu bar, 3-15
- return code, 3-15

Mouse form

- in menu bar, 1-2

Move bar, 1-5

N

NDC (see Normalized Device Coordinates), 2-7

Nil, 3-22

Normalized Device Coordinates, 2-7

- origin point, 2-7
- transformation, 2-8

Not-ready list, 2-4

O

objc_draw, 3-27

Object Library

- function summary, 1-15

OBJECT Structure, 1-10, 3-23

- getting address, 3-22
- ob_flags, 3-23
- ob_head, 3-23
- ob_height, 3-23
- ob_next, 3-23
- ob_spec, 3-23
- ob_state, 3-23
- ob_tail, 3-23
- ob_type, 3-23
- ob_width, 3-23
- ob_x, 3-23
- ob_y, 3-23

Object transformation, 3-5

Object trees, 1-11

- components, 1-11
- creating, 1-11, 3-1
- displaying, 1-11
- in RSC file, 3-2

Objects, 1-10

- illustration of types, 1-10
- index, 3-2
- index number, 1-11
- linking, 1-11
- naming, 3-2

- object tree, 1-11
- Origin point, 2-7
- OUT file, 2-10
- Output attributes, 1-18
- Output Functions, 1-18
 - attributes, 1-19
 - getting attributes, 1-20
 - setting attributes, 1-19

P

Panel

- components, 1-10
- Picture elements, 2-7
- Pixel dimensions, 2-7
 - affect, 2-8
- Pixels, 1-4, 2-7
- Porting applications, 2-9

R

- Raster Coordinates, 2-7
 - origin point, 2-7
- Raster operation functions, 1-20
- RC (see Raster Coordinates), 2-7
- RCS (see GEM RCS), 1-11
- Ready list, 2-4
- Rectangle clipping, 3-12
- Rectangle list, 3-33
- Redraw message, 3-32
- Removing fonts, 2-5
- Request mode, 1-20
- Reserved files, 2-10
- Resource file

- extension, 2-10
- Resource Library
 - function summary, 1-16
- RSC file, 2-10
- rscr_load, 3-3
- RSH file, 2-10
- rsrc_gaddr
 - getting form root address, 3-25
 - getting root dimensions, 3-22
 - menu tree, 3-8
 - object tree, 3-6

S

- Sample mode, 1-20
- SCRAP files, 2-10
- Scrap Library
 - function summary, 1-16
- Screen device
 - opening, 1-18
 - required coordinate system, 2-7
- Screen driver
 - system font, 2-5
- Screen Manager, 2-2
 - form processing, 2-3
 - user input, 2-3
- Screen redraw, 1-2
- Scroll bar, 1-5
- Shell Library
 - function summary, 1-17
- Size box, 1-5
- Slider, 1-5
 - horizontal message, 3-38
 - relationship to view area, 1-7

- setting location, 1-7
- setting size, 1-7
- vertical message, 3-38
- SYS file, 2-10
- System font, 2-5
- System memory allocation, 2-2

T

- Temporary memory, 2-5
- Text attributes, 3-18
- Titles (in menus), 1-2
 - conversion, 1-2
 - resetting, 3-20
- Title bar, 1-4
 - contents, 1-4
 - length, 1-4

U

- undo_mfdb, 3-30
 - size, 3-5
- User input, 1-13

V

- v_clswk, 3-41
- v_gtext, 3-19
- v_opnvwk, 3-3
- v_pline, 3-14
- VDI, 1-17
 - Attribute functions, 1-19
 - compensating for aspect ratio, 2-8
 - Escape functions, 1-20

- Input functions, 1-20
- Inquire functions, 1-20
- Output functions, 1-18
- Raster operation functions, 1-20
 - selecting coordinate system, 2-7
 - transforming coordinates, 2-8
- Workstation Control functions, 1-18
- View Area, 1-7
 - changing, 3-37
 - related messages, 3-37
 - scroll bar and slider, 1-7
 - size of, 1-7
- Virtual Device Interface, 1-17
- Virtual workstation
 - opening, 1-18
- vr_trnfm, 3-6
- vro_copyfm, 3-30
- vro_cpyfm, 3-32
 - updating view area, 3-38
- vsf_color, 3-13
- vsf_interior, 3-13
- vsl_color, 3-13
- vsl_type, 3-13
- vst_alignment, 3-18
- vst_color, 3-18
- vst_height, 3-18
 - return values, 3-18
- vst_point, 3-19
- vswr_mode, 3-13, 3-18
 - for text, 3-18

W

- wind_calc, 3-35
- wind_close, 3-41
- wind_create, 3-8
- wind_delete, 3-41
- wind_get, 3-33
 - current window, 3-36
 - first rectangle, 3-33
 - full window, 3-36
 - getting desktop window work area, 3-8
 - next rectangle, 3-33
 - previous window, 3-36
 - rectangle list, 3-33
 - WF_CXYWH, 3-36
 - WF_FIRSTWXYWH, 3-33
 - WF_FXYWH, 3-36
 - WF_NEXTWXYWH, 3-33
 - WF_PXYWH, 3-36
 - work area of current window, 3-39
- wind_set
 - changing window location, 3-35
 - changing window size, 3-35
 - current window, 3-35
 - set title bar, 3-8
 - setting horizontal slider location, 3-39
 - setting horizontal slider size, 3-39
 - setting top window, 3-17
 - setting vertical slider location, 3-39
 - setting vertical slider size, 3-39
- WF_CXYWH, 3-35
- WF_HSLIDE, 3-39
- WF_HSLSIZ, 3-39
- WF_VSLIDE, 3-39
- WF_VSLSIZ, 3-39
- wind_update
 - main event loop, 3-10
 - taking control of window, 3-3
- Window, 1-2
- Window Control Areas, 1-5
 - arrows, 1-5
 - close box, 1-5
 - customary responses, 1-5
 - full box, 1-5
 - move bar, 1-5
 - scroll bar and slider, 1-5
 - selecting, 1-4, 3-8
 - size box, 1-5
- Window Library, 1-4
 - function summary, 1-16
- WM_ARROWED, 3-38
- WM_CLOSED, 3-16
- WM_FULLED, 3-16, 3-36
- WM_HSLID, 3-38
- WM_MOVED, 3-35
- WM_REDRAW, 3-16
- WM_REDRAW
 - data, 3-32
- WM_SIZED, 3-35
- WM_TOPPED, 3-16
- WM_VSLID, 3-38
- Work area
 - desktop window, 1-3
 - memory form, 3-30
 - moving, 3-35
 - output functions, 1-7
 - rectangle intersection, 3-33

- redrawing, 3-32
- saving screen, 3-30
- size and location, 1-7
- size of, 1-4
- sizing, 3-35
- switching, 3-35
- user input management, 1-7
- Workstation, 1-18**
 - getting characteristics, 1-18
 - handle, 1-18
 - opening, 1-18
 - setting attributes, 1-18
- Workstation control functions, 1-18**
- World coordinate space, 1-7**
 - in DEMO, 3-5
 - view area, 1-7
- Writing mode, 1-19**
 - options, 3-12

