COMPUTE!'s
Technical Reference Guide

ATARI ST

VOLUME TWO

AES

Sheldon Leemon

A practical tutorial and reference to the GEM AES. Includes program examples in C, BASIC, and machine language. For the intermediate to advanced Atari ST programmer.

A **COMPUTE! Books** Publication

COMPUTE!'s Technical Reference Guide

ATARI ST

VOLUME TWO

GEM AES

Sheldon Leemon

COMPUTE! Publications, Inc. abc
A Capital Cities/ABC, Inc. Company
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-114-5

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a Capital Cities/ABC, Inc. company and is not associated with any manufacturer of personal computers. Atari, ST, ST BASIC, 520ST, 1040ST, and TOS are trademarks or registered trademarks of Atari Corporation. GEM is a trademark of Digital Research, Inc.

Contents

Foreword

This is the second book in a series of three on the Atari ST. The first concerned the VDI (the Virtual Device Interface). *COMPUTE!'s Technical Reference Guide—Atari ST, Volume Two: The GEM AES* takes you further into the underpinnings of the ST's fast, friendly GEM interface with a thorough examination of the Application Environment Services (AES).

This book will help you to understand the AES. Using the AES will enable you to make your ST programs as friendly and cooperative with the user as the intuitive GEM environment can make them.

COMPUTE!'s Technical Reference Guide—Atari ST, Volume Two: The GEM AES explains ST multitasking, window operation, dialog boxes, alert boxes, menus, input, and output. The latter half of this book is a complete reference to AES functions listed according to opcode, along with an alphabetical index. These functions are complete with C bindings, an explanation of input and results, and more.

In each chapter, functions are covered individually in detail with examples of their use, culminating in fully commented sample programs written in C, machine language, and BASIC.

If you are an ST programmer, this is the reference you have been looking for.

Chapter 1

GEM and the AES

Although many people think of GEM as a computer operating system, Digital Research, Inc. (DRI), the creators of GEM, prefer to call it an operating *environment.* Traditional microcomputer operating systems provide access to the most basic I/O (Input/Output) devices, like the keyboard, display screen, and disk drives, but they only support the transfer of text characters. Newer microcomputers, however, offer a more sophisticated class of input/output operations, which are commonly grouped together under the term *graphics interface.* Computers that offer such an interface allow users to operate programs by manipulating graphics objects on the display screen with a pointing device known as a mouse, rather than requiring them to type in carefully worded commands.

GEM (an acronym for Graphics Environment Manager) was designed to provide this graphics interface. An additional level of system routines, it sits on top of the low-level input/output functions furnished by the computer's own operating system. Because GEM supplements, rather than replaces, the existing operating system, it's possible to write "old-style" computer programs on the ST that take text input from the keyboard, without using GEM at all. These are called TOS (Tramiel Operating System) programs. TOS programs don't take full advantage of the capabilities of the ST.

GEM programs are easier to operate, because of their visual orientation—as the old saying goes: One picture is worth a thousand words. Anyone who's ever taken out the garbage should be able to understand the effect of dragging a file on the Desktop. Moreover, GEM provides a consistent context for program operation. Users of GEM programs who want to learn what the various program options are, know that they can always find the menu selection on the menu bar at the top of the screen. Other standard features like dialog boxes, alert boxes, icons, and file selectors provide a comfortable frame of reference even when you're using programs you've never seen before.

From the programmer's standpoint, GEM makes it easy to provide the user with such advanced features as overlapping windows, drop-down menus, and icons. These features would be difficult for programmers to implement without the GEM environment. GEM also offers portability for programs among the various versions of the ST and also limited portablility to the IBM PC and compatibles. GEM also includes facilities to provide output without regard to the device-specific resolution. This is of particular interest now that Atari has announced plans to introduce a laser printer.

The programmer should keep in mind, however, that portability has its price. A more generalized program will always be slower and less efficient than a more specific one. Although the ST has a fast and powerful processor, its operations will slow down if GEM has to do a lot of internal coordinate conversion and range checking. Programmers may find that in some cases they must use more device-specific methods to achieve the desired level of performance. While portability is nice, you'll have to decide what performance compromises are acceptable in order to gain its benefits.

GEM Organization

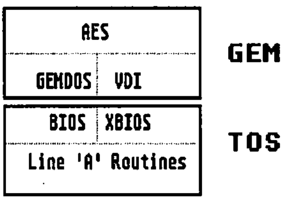
Though GEM is spoken of as a single entity, it's actually made up of a number of parts. The two major divisions are the Virtual Device Interface (VDI) and the Application Environment Services (AES).

The VDI was discussed in Volume One of this series. It provides a number of low-level, device-independent drawing routines, also known as *graphics primitives.* It also supplies some fundamental input functions for receiving data from the mouse and keyboard.

The AES provides user-interface features called *environment services*. These are the features usually associated with GEM: drop-down menus, overlapping windows, icons, and dialog boxes.

It's important to understand that ST system software is hierarchical (see Figure 1-1). At the bottom of the hierarchy are the routines known as the BIOS (Basic Input/Output System), XBIOS (eXtended BIOS) and Line A (graphics) routines. These routines communicate directly with the ST hardware and peripherals. The GEMDOS (Disk Operating System) builds on the disk-access routines in the XBIOS to provide a filing system. Similarly, the GEM VDI builds upon low-level Line A graphics routines to provide higher-level, device-independent graphics routines. Finally, the GEM AES uses the graphics and input primitives supplied by the VDI to provide a sophisticated user interface.

Figure 1-1. Parts of the ST Operating System



The AES is itself made up of a number of parts. At the lowest level is the multitasking *kernel*. Its job is to provide GEM with a limited form of multitasking. Multitasking allows processing time to be divided between the primary application, the desk accessory programs, and the AES Screen Manager (discussed below), so that they all appear to be running at the same time. Of course, the ST's 68000 microprocessor can't execute instructions for more than one task at a time, but because it operates at a high rate of speed, it's possible to switch between tasks quickly enough to give the user the impression that they're running simultaneously.

The scheduling system for tasks is fairly simple. The kernel maintains two lists of tasks, the *Ready* list, and the *Not-Ready* list. In order to understand the difference between the two, you must first know something about GEM events. In single-tasking microcomputer systems, a program finds out about an I/O event—such as the user pressing a key on the keyboard—by continuously checking the status of the I/O device until a specified event occurs. The processor is on hold, waiting until it receives input.

Multitasking systems use a more efficient process. They suspend a task that's waiting for some I/O event by placing it on the Not-Ready list. Then the other tasks take turns running until the event occurs.

GEM provides a number of system calls for just this purpose. These calls are part of the Event Library, and they allow an application to wait for a keypress, mouse button press, mouse movement, timer alarm, and/or a message from another task. Tasks that use these calls to indicate they're waiting for an I/O event are put on the Not-Ready list. They're inactive until the event they are waiting for occurs. All the tasks that are ready to run are kept on the Ready list, where they take turns executing.

The task currently running is at the top of the Ready list. When that task makes a call to one of the AES library routines, the Dispatcher portion of the kernel takes control as soon as the call is completed. If the call was an event call, indicating that this task wants to wait for an I/O event, the Dispatcher moves the task from the Ready list to the Not-Ready list. Otherwise, it merely terminates execution of the task, moves it from the top of the Ready list to the bottom, and moves all of the rest of the tasks up one place on the list. Control is handed over to the new task at the head of the list, and it's allowed to run until it makes an AES call.

Unlike some multitasking systems, where each task gets a fixed amount of time to execute, the AES dispatcher will not preempt a running task. The only way it can get control is for the task to make an AES call. Therefore, if your application goes for a long time without making an AES call (during extensive math calculations, for example), multitasking will break down. The Screen Manager task will not run, and the GEM features that it handles—such as the menu bar—will no longer work, puzzling the user. Therefore, it's recommended that your program make periodic AES calls, if only to keep the dispatcher working. One "harmless" AES call that you can make is to evnt_timer( ), specifying that your application wishes to wait for a period of zero milliseconds.

While tasks rotate through the Ready list in a fixed order, tasks on the Not-Ready list aren't arranged in any particular sequence. When an awaited I/O event occurs, the task waiting for that event is removed from the Not-Ready list and added to the bottom of the Ready list.

Another vital part of the AES is the Screen Manager. The Screen Manager is a separate GEM task that shares processor time with the main application program.

To understand its function, you must first realize that many of the user-interface features of GEM require cooperation between GEM and the application program in order to work. Although most GEM applications share common user-interface features, such as menus and dialog boxes, these features prompt a unique response from each application. GEM therefore provides for a division of labor between the AES and the application. Since only the application knows what it wants to display, it alone is held responsible for everything that goes on in the active (topmost) window.

The Screen Manager's job is to monitor mouse movements outside of the active window, provide visual feedback where appropriate, and report the outcome of the user's activity when that activity has some meaning to the application.

For example, say the user moves the mouse to the size box at the bottom right corner of a window border. When the mouse pointer moves out of the active area of the window and into the window border, the Screen Manager watches it. If the user presses the left mouse button while the pointer is over the size box and holds it down while moving the mouse, the Screen Manager is responsible for drawing a dotted window outline that follows the mouse pointer. When the user releases the mouse button, the Screen Manager sends a message to the application telling it the size of the window requested, and control returns to the application. Then it's up to the program to decide whether to change the size of the window in compliance with the user's request.

The Screen Manager's major areas of responsibility are handling the drop-down menu system and the window controls. When the mouse pointer crosses into the menu bar, the Screen Manager saves the screen rectangle where a menu is to be displayed, and then it displays the menu. It handles the highlighting of menu items as the pointer travels over them, and redisplays the saved screen area if the left mouse button is clicked. If the button was clicked while the pointer was over a menu item, the Screen Manager sends a message to the application, specifying the menu and item numbers selected. Similarly, when the user clicks on one of the window controls, such as the closer (close box), the fuller (full box), the sizer (size box), scroll bars, or arrows, the Screen Manager sends the appropriate message to the application. If the user drags the window's drag bar, the Screen Manager may be able to redraw the window in the new position itself, so long as the visible portion of the window remains the same. The Screen Manager will also notify the application if the user clicks in an inactive window, signaling that he or she wishes to make it the active one.

While the multitasking kernel and the Screen Manager are necessary to the functioning of GEM applications, they do their work "behind the scenes." The programmer doesn't communicate directly with them.

The portion of the AES that's most accessible to the programmer, and therefore the part dealt with here in greatest detail, is known as the AES Libraries. These are collections of operating system calls that perform tasks related to the setup and operation of the user interface. There are eleven different libraries in all. Their names and functions are as follows:

Application Library. This library contains routines to register an application with GEM, and to send messages back and forth between tasks.

Event Library. The functions found here allow you to put the application on the Not-Ready list until certain I/O events occur (mouse button press, mouse movement, keypress, timer expiration, or message events) and to pass information about the events back to the application, once they do occur.

Menu Library. The routines in this library allow the application to establish and maintain drop-down menus.

Object Library. This library contains routines that allow the program to interact with various types of GEM *objects*. GEM objects are data structures that form the basic building blocks of icons, menus, dialog boxes, and alert boxes.

Form Library. This library contains routines that allow the application to display and handle dialog and alert boxes.

Graphics Library. The routines in this library perform some graphics functions (mainly related to managing box outlines as they appear on the Desktop), and some low-level I/O functions.

Scrap Library. The functions in this library are used to read and write data to *clipboard* files on disk. Clipboard files store data for interchange among applications.

File Selector Library. This library contains a single routine that displays and manages the standard file-selection dialog.

Window Library. This library contains routines that support the creation and management of overlapping windows.

Resource Library. The functions in this library enable the application to load and use the object information stored in resource (.RSC) files. This information is used to display menus, dialog boxes, and so on.

Shell Library. The routines found here deal with loading and running programs. The Shell Library allows you to load and run other applications from within an application (like the Desktop does).

At this point, you may be wondering where the GEM Desktop fits into the scheme of things. Strange as it seems, the Desktop is not really an integral part of the GEM AES. Rather, it is itself an application program, and as such, it uses GEM the same way your application programs will. Of course, you should bear in mind that most, if not all, of the AES Library functions were originally written to facilitate creation of the Desktop application. Although most of these library calls are general enough to be of use in almost any application, a few are of interest mainly for their use in the Desktop.

Allocating RAM

The parts of the AES discussed so far consist of program code located in the ST's operating system ROM. The AES also uses some free system RAM.

The first RAM area it must allocate is the Menu/Alert Buffer. As was mentioned above, when the user pulls down a menu, the Screen Manager automatically saves the portion of the screen that lies under the menu. The Menu/Alert Buffer is the place where this screen information is stored. This buffer has to be large enough to accommodate a quarter of the screen at any one time. Since the ST uses 32K of screen memory in all of its resolution modes, the Menu/Alert Buffer must occupy 8K of RAM.

AES may also need some RAM to load resource files. Resource files are data files which define GEM object trees, the data structures used to create menus, dialog boxes, and icons. The amount of RAM required depends on the size of the resource files to be loaded.

Finally, GEM may also need to use some free RAM for the Desk Accessory Buffer. This is the area used to store up to six desk accessory programs at boot time. Desk accessories are separate special programs. They can be found on the root directory of the startup disk. Their file names end in ".ACC".

These programs may be run from the "Desk" menu of another application.

Using the AES Libraries

You may think of the AES Libraries as a collection of subroutines that you can call from your program. In order to pass data to these subroutines and receive data from them in return, you must allocate storage space in computer memory for a number of data arrays. The AES uses information from six different arrays, each of which is made up of a number of 16-bit (two-byte) values. These arrays are as follows:

Array Name	Size	Function
global	15 words	Global parameters
control	5 words	Control parameters
int_in	16 words	Input parameter
int_out	7 words	Output parameters
addr_in	2 long words	Input addresses
addr_out	1 long word	Output addresses

The array named global contains certain information about GEM and the application which must be available to all of the library routines. The array contains nine elements, the first three of which consist of two-byte words. The rest are four-byte long words. These elements are as follows:

Address	Element	Name	Contents
global	global(0)	ap_version	The GEM AES version number
global+2	global(1)	ap_count	The maximum number of concurrent applications supported by this GEM version
global+4	global(2)	ap_id	A unique ID number for the currently active application (used to pass messages to it)
global+6	global(3,4)	ap_private	A private storage place to be used by the application
global+10	global(5,6)	ap_ptree	A pointer to the address of the header of the object tree loaded with rsrc_load( )
global+14	global(7,8)	ap_1resv	Reserved for future use (0)
global+18	global(9,10)	ap_2resv	Reserved for future use (0)
global+22	global(11,12)	ap_3resv	Reserved for future use (0)
global+26	global(13,14)	ap_4 resv	Reserved for future use (0)

As you can see, there are official GEM names for each of these elements. The first, ap_version, is an internal GEM AES version number, supplied by GEM after an application is initialized with the appl_init( ) call. The version in use at the time of this writing was 288. The second, ap_count, is also returned by GEM, and it shows how many applications can run concurrently under this version of GEM. Since the current ST does not support full multitasking, only one main application can run at a time. The third element, ap_id, is where GEM stores the unique application ID number that identifies this application. The ID number can be used when passing messages to this application from another task (such as a desk accessory). This ID number is the same number that is supposed to be returned by the appl_init( ) call, but some versions of the C language bindings don't correctly return the ID number. Since appl_init( ) does place the ID number in the global array, an application can find it out by reading that array directly. The C library assigns the name gl_apid to the ap_id element, so a C application can find out its value by declaring gl_apid as an external int and using the contents of that variable.

The rest of the global array is made up of 32-bit long words. The ap_private element is reserved for use by the application, any four bytes of data, as determined by the programmer, may be stored here. The ap_ptree element is where the AES stores a pointer to the header of the object tree loaded with the rsrc_load( ) call. The other four elements are reserved for future use.

The second data array used by the AES is named control, and it consists of five elements, each two bytes in length. The information stored in each of these elements is as follows:

Address	Element	Control Parameter
control	control(0)	Command opcode (operation code)
control+2	control(1)	Number of integer inputs passed in int_in
control+4	control(2)	Number of integer results returned in int_out
control+6	control(3)	Number of input addresses passed in addr_in
control+8	control(4)	Number of addresses returned in addr_out

The first element of the control array is used to pass the opcode. Since all of the AES routines have a common entry point, there has to be some way to tell the AES which command is to be executed. Therefore, each command is given an identification number called an opcode. These opcodes are grouped by library. The Application Library uses opcodes 10–19, the Event Library uses opcodes 20–26, and so on.

The remaining four elements are used to indicate how an AES call utilizes the parameter arrays int_in, int_out, addr_in, and addr_out. The two input parameter arrays, int_in and addr_in, are used by the application to pass values to the AES library calls. These arrays give the function call information about how the application wants the calls to operate. The two output parameter arrays, int_out and addr_out, are used by the AES library to return results to the application. The last four control array elements are used to indicate how many parameters are being passed in each direction. The application uses control(1) to specify the number of input integers being passed in the int_in array and control(3) to indicate the number of addresses being passed in addr_in. The AES uses control(2) to specify the number of integers being returned in int_out and control(4) to specify the number of addresses it has returned in addr_out.

Machine Language AES Calls

If you are programming at the machine-language level, you must explicitly reserve memory space for each of these arrays and put the proper values in each of the memory locations before calling the command. The first step is reserving space for each of the data arrays:

Array Name	**Storage Space**
global:	.ds.w 15
control:	.ds.w 5
int_in:	.ds.w 16
addr_in	.ds.l 2
int_out:	.ds.w 7
addr_out:	.ds.l 1

Since each AES call uses a fixed number of inputs and outputs, it's possible to determine the maximum number of bytes need for these arrays. In addition to allocating data-array space, you must also define an AES parameter block. This parameter block contains the beginning address of each of the six data arrays:

```
apb:  .dc.l control,global,int_in,int_out,addr_in,addr_out
```

These addresses must be arranged in the order shown above, since the AES uses the parameter block to find the data arrays. Once you've set up the arrays and the parameter block, you must place any input parameters into their correct place in the data arrays. For example, to execute the graf_mouse( ) command to change the shape of the mouse pointer, you would transfer the following values:

```
move #78,control     ;Move the graf_mouse opcode (78) to
                        control(0)
move #1,control+2    ;Move the length of int_in array (1) to
                        control(1)
move #1,control+4    ;Move the length of int_out array (1) to
                        control(2)
move #1,control+6    ;Move the length of addr_in array (1) to
                        control(3)
move #0,control+8    ;Move the length of addr_out array (0) to
                        control(4)
move #3,int_in       ;Move the mouse pointer shape code to
                        int_in(0)
```

Now you're ready to call the AES. First, place the address of the AES parameter block into register d1. Next, move the AES identifier code (200 or $C8) into register d0. Finally, call the AES with a "trap 2" instruction. This initiates a software-generated exception (similar to a hardware interrupt) that causes execution of an exception-handler routine. In this case, the routine executed is the one whose address is pointed to by the long word beginning at location 136 ($88). This routine is the one used to handle all GEM VDI and AES calls (VDI calls are identified by placing a value of 115 or $73 into register d0). The instruction sequence used for making a AES call looks like this:

```
move.l  #apb,d1   ;Move address of AES parameter block to d1
move.l  #$C8,d0   ;Move AES identifier ($C8) into d0
trap    #2        ;call GEM entry point
```

Please note that the procedures outlined above just cover the steps required to make the AES call itself. Before you get to that stage, you must take preparatory steps to set up the program environment (for instance, allocating stack space) and the graphics environment (opening a GEM output workstation). This will be outlined in the next chapter and illustrated in an example program.

ST BASIC VDI Calls

Making ST BASIC calls employs the same fundamental strategy for AES calls as making calls from machine language programs. The only difference is that BASIC does much of the preparatory work for you. Since the BASIC interpreter itself must use AES calls, it already has set aside memory for the data arrays control, global, int_in, int_out, addr_in, and addr_out, and has set up a parameter block with the starting address of each of these arrays. BASIC assigns the reserved variable GB (for *GEM Base*) to the address of the AES Parameter Block. This means that the starting address of the control array can be found by using the PEEK command to obtain the address stored at the four bytes starting with address GB. In order to PEEK a four-byte number in ST BASIC, you must specify the address to PEEK as a double-precision number. This can be done by assigning the value in GB to a variable that has been declared to be double-precision:

```
10 apb# = gb
20 control = PEEK(apb#)
```

The pound sign at the end of apb# tells BASIC that apb# is a double-precision variable. Since each address found in the Parameter Block is four bytes long, you can find the address of each succeeding data array by PEEKing the next four bytes in memory:

```
30 global = PEEK(apb# + 4)
40 gintin = PEEK(apb# +8)
50 gintout = PEEK(apb# +12)
60 addrin# = PEEK(apb# + 16)
70 addrout# = PEEK(apb# + 20)
```

Notice that we used the variable names gintin and gintout for the int_in and int_out arrays. That's because the original version of ST BASIC doesn't allow the underscore character in variable names, and the names intin and intout are already reserved for the data arrays used by the GEM VDI. Notice also that we've placed a pound sign at the end of the variables that hold the addresses of addr_in and addr_out. That's because the values in these arrays are four bytes long. By declaring the type of these variables as double-precision, BASIC will know it should PEEK or POKE four bytes at a time.

You may read the contents of the various data arrays, or write to them, by using the PEEK and POKE commands. Since each element in gintin and gintout is two bytes long, you must multiply the element number by 2 to get the proper offset for the POKE statement. Each element in addrin# and addrout# is four bytes long, so you must multiply the element number by 4 to get the proper offset for those arrays. To write a value to int_out(1), you'd POKE gintout+4, and to write a value to addr_in(3), POKE addrin+12. The following short program shows how to change the shape of the mouse pointer from an arrow to a pointing hand with the graf_mouse( ) call from BASIC.

```
10 apb# = gb
20 control = PEEK(apb#)
30 global = PEEK(apb# + 4)
40 gintin = PEEK(apb# +8)
50 gintout = PEEK(apb# +12)
60 addrin = PEEK(apb# + 16)
70 addrout = PEEK(apb# + 20)
80 POKE gintin,3: REM Pointing hand is shape number 3
90 GEMSYS(78) : REM call graf_mouse( )
```

This method of making an AES call from BASIC is similar to that used in the machine language program shown above, in that the input parameters are placed directly in the int_in array. But you'll also notice that the GEMSYS call takes care of a lot of the detail work. First, there was no need to POKE a value for the opcode into control(0), because the opcode is passed as part of the GEMSYS call. Second, the GEMSYS command performs the same tasks as the three lines of machine language code: It places the address of the parameter block into register d1, places the AES identifier code into d0, and then executes the TRAP #2 statement.

The original version of ST BASIC doesn't contain any built-in commands that perform the same functions as AES calls. Although not yet released at the time of this writing, the revised MCC BASIC promises to include a few, such as ASK

MOUSE, which returns the current mouse position like graf_mkstate( ). And the revised BASIC is slated to include reserved variables for the data arrays, such as GEM_ADDRIN, GEM_ADDROUT, GEM_CONTRL, GEM_GLOBAL, GEM_INTIN, GEM_INTOUT, and a STATUS variable to return information from AES calls. Even so, BASIC programmers will still have to learn the details of making AES calls if they want to take full advantage of GEM.

Calling the AES Routines from C

It's much easier to make AES function calls from C than from either machine language or BASIC. That's because C compiler packages for the ST include one or more function libraries known as GEM *bindings*. These bindings are object-code library files that define a separate named function for each GEM call. When the C program is linked to the proper library files, it can call GEM functions as if they were part of the C language.

These library files also allocate storage space for the data arrays. But the programmer is not responsible for placing data directly into these arrays. Instead, input parameters are passed to the binding functions as part of the function call. For example, you could execute the graf_mouse( ) command performed by the BASIC and machine language programs like this:

```
int dummy, shape = 3;
graf_mouse(shape,&dummy);
```

The function defined as graf_mouse( ) in the library takes the parameter *shape* that's passed to it, puts it in int_in(0), and puts the address of the parameter *dummy* in addr_in(0). It also puts a 1 in control(1), control(2), and control(3), a 0 in control(4), and places the command opcode (78) in control(0). The function graf_mouse then loads registers d0 and d1 with the proper values and executes a TRAP #2 instruction. In short, it takes over all of the repetitive steps associated with making GEM calls, allowing the programmer to concentrate on the essential aspects of the function.

Making GEM calls from C is easy. It's because of this, and because C programs are relatively small and quick (compared to other high-level languages), C has become the language of choice for software development on the ST. Most of the examples in this book are written in C. On occasion, however, machine language and BASIC examples will be included as well, to show how the C examples could be translated. The C function names will be used as they appear in the official Digital Research GEM bindings, since they have been adopted by the manufacturers of most other C compilers as well. Refer to the user's manual of your particular C compiler for specific information concerning the C function names.

The C programs in this book are designed to work specifically with the *Alcyon C* compiler, the one officially supported by Atari, and with *Megamax C,* which also provides a very complete development environment. For these compilers, the integer data type *(int)* refers to a 16-bit word of data. Other compilers, such as *Lattice C,* use a 32-bit integer as the default data type. When compiling the programs in this book with such compilers, you should substitute the word *short* for each reference to int. For the sake of simplicity, we have not used the portability macros such as WORD, which use the C preprocessor to define a 16-bit data type that will be valid for any compiler, but you are free to do so.

Chapter 2

Starting an Application: Windows, Part 1

The first step in starting a GEM application is to initialize it with a call to the AES Application Library routine *appl_init( )*. This call registers the application with the AES, which then initializes several items in the global data array. One of these is an application ID number which the AES assigns to the application. This ID number is used by other tasks (the GEM Screen Manager, for instance) when they wish to communicate with the application through its message buffer. The C format for the call is

```
int ap_id;
ap_id = appl_init( );
```

where *ap_id* is supposed to be the application ID number. Note, however, that as of this writing, the bindings for all C compilers which derive from source code supplied by Atari (such as *Alcyon* and *Megamax C*), do not correctly return the application ID number. Instead, these bindings always return a value of 1 in the variable ap_id. The actual ID number is, however, correctly stored in the third element of the global array. This element is assigned the variable name gl_apid by the bindings. If your C compiler library does not return the correct value, you can work around this bug by declaring the external variable *gl_apid* and getting the value from this variable:

```
extern int gl_apid;
int ap_id;
appl_int( );
ap_id = gl_apid;
```

If you've registered an application with the AES, be sure to "unregister" it before your application terminates. You perform this function with a call to *appl_exit( )*; as follows:

```
int status;
status = appl_exit( );
```

Status is an error return code. A value of 0 in status indicates that an error occurred, while a positive integer value indicates that the application exited successfully.

Opening a Virtual Screen Workstation

Since most GEM programs use at least some of the VDI graphics function, you'll need to prepare a graphics output environment by opening a VDI screen workstation. Though this process is covered in detail in Volume One of this series, a brief review follows. The VDI call used to open a graphics workstation is *v_opnvwk( );*. It's invoked like this:

```
int input[12];
int output[57]
int handle;

v_opnvwk(input, &handle, output)
```

The *input* array consists of 12 words of data passed to the VDI to specify the initial default graphics settings for the workstation. With two exceptions, you can set these to a default value of 1. The first of these exceptions is input[10], which is used to select the graphics coordinate system. Initialize this value to 2, indicating that you wish to use raster coordinates, which correspond to the ST's actual screen dimensions. The second exception is input[0], the device ID number. On ST systems that have the GDOS extensions loaded, this device number specifies the screen resolution mode. To find the resolution mode, use XBIOS command 4. C programmers can use *getrez,* a macro defined in the file osbinds.h to call this function. Getrez returns the numbers 0 for lo res, 1 for medium res, and 2 for high res. To get the proper screen device number, you must add 2 to the value returned by getrez:

```
#include <osbinds.h>

int rez, work_in[12];

rez = getrez( )
work_in[0]= rez + 2;
```

Machine language programmers can perform the getrez call using the following code:

```
move.w   #4,-(sp)    * push command number on stack
trap     #14         * call XBIOS
addq.l   #2,sp       * pop command number off the stack
```

The resolution will be returned in register d0.

The other input value is &handle. This is a pointer to the variable that holds the physical workstation ID number of the screen device. In order to discover this workstation handle, the AES Graphics Library function, *graf_handle( )* must be used. The format for this call is

```
int phys_handle, cellw, cellh, chboxw, chboxh;
phys_handle = graf_handle(&cellw, &cellh, &boxw, &boxh);
```

The physical screen handle is returned in *phys_handle.* Before making the v_opnvwk call, phys_handle should be stored in the variable handle. In addition to the the physical screen handle, graf_handle( ) returns some interesting information about the size of the default-system text font used in menus and dialog boxes. The width and height of the character cell are returned in *cellw* and *cellh.* The character cell is the entire space taken up by each character, including the intercharacter spacing. The width and height of a box that surrounds the text cell are returned in *chboxw* and *chboxh.* The chboxw and chboxh measurements are significant because many GEM Objects are scaled to these sizes. For example, the window controls such as the title bar, close box, size box, and scroll bars are chboxw wide and chboxh tall.

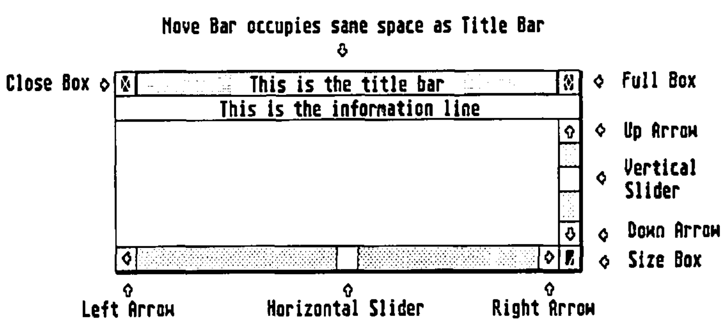
Opening a Window

The next step in setting up an application is to open an output window. Windows are an integral part of the GEM user interface. They are used to divide a single display screen into separate, sometimes overlapping, sections, each surrounded by a visible border. This makes it possible for a single application to present several distinct types of information on the same screen. For example, a program could display help information in one window and data in another. Windows also facilitate sharing the display between the application and one or more desk accessories.

There are two types of GEM windows, the Desktop window and application windows. The Desktop window is controlled by the GEM AES, and is present when any GEM application (such as the Desktop program) is running. It covers the entire display area and is divided into two parts, the menu bar and the Desktop work area. The menu bar stretches across the top line of the screen and is the same height as a character cell in the default text font. When a GEM application first starts, the menu bar contains the name of the application file. The application may request the AES to replace this filename with a drop-down menu, as will be seen in a later chapter. The Desktop work area covers the rest of the display space. It provides a background for windows which the application opens. The default background appears as a field of light green on a color system and gray on a monochrome system.

Application windows are opened and controlled by the application. In addition to a border, these windows may contain a number of different components, located in and around the window borders (see Figure 2-1). Most of these allow the user to control certain aspects of the window's appearance and function. These control features include a title bar, a move bar, an information line, a close box, a full box, and horizontal and/or vertical slide bars.

Figure 2-1. The Component Parts of a GEM Window



Title bar. The title bar stretches across the top of the window and is used to display the name of the window.

Move bar. The move bar occupies the same area as the title bar and has no visible characteristics to distinguish it from the title bar. The move bar allows the user to move the window around on the screen.

Information line. This line also stretches across the top of the window, directly below the title bar. It is used by the program to display additional information.

Close box. This is a box the size of a single character, located at the left side of the title bar. It allows the user to close the window.

Full box. This is a single character located in the right corner of the title bar, which allows the user to expand the window to full-screen size or to retract it to its original size and position.

Horizontal and/or vertical slide bars. These bars stretch across the bottom or right window borders. They contain a rectangle called a slider with optional arrows at either end. The arrows are used to move through the contents of the windows a single character at a time, while the slider is used to scroll the window contents in arbitrary increments. It is also possible to move through the window a page at a time by clicking on the part of the bar located between the slider and the arrows.

Note that in most cases, the AES itself does not respond to the user's request. For example, the AES does not close the window when the user clicks on the close box nor size it when the user drags the size box. Instead, it sends a message to the program, notifying it of the user's actions. The AES message system will be discussed in Chapter 3.

The rest of the window—the area inside the borders—is the application's work space. Windows appear to divide the screen into separate areas, but these divisions are only logical constructs, not physical fact. GEM provides the framework for windowing, but it's up to the AES and the application program to actually manage the windows and their contents. The AES is responsible for drawing and maintaining the window borders and the controls placed within them. The application is responsible for everything that goes on inside the work space. The program must keep track of the size and position of the active (topmost) window, and make certain that it restricts its output to the confines of the work area. Otherwise, the window's borders would be no more substantial than lines drawn on the screen.

Window Creation

The first step in displaying a window is to define its maximum size and composition with the Window Library call *wind_create( ).* This function causes the AES to allocate a window and initialize the data that the AES uses to keep track of the window. It doesn't display the window. The format for the wind_create( ) call is

```
int wi_handle, controls, fullx, fully, fullw, fullh;
wi_handle = wind_create(controls, fullx, fully, fullw, fullh);
```

where *controls* is a bit flag which tells the AES which of the 12 window controls to attach to the window. Each of the window controls is assigned a bit. If the bit that represents a particular window control is set to 1, that control will be attached to the window. The 12 possible window attributes are as follows:

Bit	Bit Value	Macro Name	Window Control
0	1 (0x001)	NAME	Title bar
1	2 (0x002)	CLOSER	Close box
2	4 (0x004)	FULLER	Full box
3	8 (0x008)	MOVER	Move bar
4	16 (0x010)	INFO	Information line
5	32 (0x020)	SIZER	Size box
6	64 (0x040)	UPARROW	Up-arrow for vertical scroll bar
7	128 (0x080)	DNARROW	Down-arrow for vertical scroll bar
8	256 (0x100)	VSLIDE	Slider for vertical scroll bar
9	512 (0x200)	LFARROW	Left arrow for horizontal scroll bar
10	1024 (0x400)	RTARROW	Right-arrow for horizontal scroll bar
11	2048 (0x800)	HSLIDE	Slider for horizontal scroll bar

Since each control is represented by a separate bit, any or all of the controls can be attached to a given window. To create a window that has a title bar, a close box, and a size box, you would set control to a value of 35 (1 + 2 + 32). C programmers can take advantage of the fact that the header file GEMDEFS.H which comes supplied with most C compilers contains macro definitions that give names to each of the controls. If your program starts with the directive #include <gemdefs.h>, you could create a window with the same set of controls specified above by setting controls to (NAME | CLOSER | SIZER). This expression still equals 35, but it's much easier to understand what the window attributes are when you look at it.

The other input parameters you must pass to wind_create( ) are the window's maximum dimensions. This isn't necessarily the size to which the window will be drawn—that's determined by the *wind_open( )* call, as you'll see. Since a window's size may be changed under program control, this set of parameters is used to specify the largest possible size to which it may be changed. This is usually equal to the size of the Desktop window's work area, that is, all of the display area except for the menu bar at the top of the screen. The procedure for finding out the dimensions of the Desktop window's work area will be discussed below. Note, however, that the AES doesn't limit your window size to the size of the Desktop work area. It's possible, for example, to create a window whose work area fills the entire display screen. Such a window will have no visible distinguishing marks, since the border and control boxes are out of the range of display memory and cannot be drawn.

The wind_create( ) call uses the standard AES system for specifying the size and position of a window or other rectangle. The four values used to delimit the rectangle are its horizontal (x) coordinate, vertical (y) coordinate, width, and height. Those of you who are familiar with the VDI will notice that this system is different from the one used to describe a rectangle to the VDI. In that system, a pointer to an array holding the x and y coordinates for two opposite corners of the rectangle is used. For example, you might describe a VDI rectangle by specifying that its top left corner is at coordinate 10,10, and its bottom right corner is at 109,109. For purposes of the AES, you would describe that same rectangle by saying that its origin is at 10,10, and its width and height are each 100 units. Each system has its advantages. The VDI system may be a bit quicker to use for drawing, since all of the coordinates are spelled out. The AES system is more flexible, however, since to move the rectangle you need only change the coordinates of the point of origin. To the programmer, the most important aspect of the difference between the two systems is that some translation is required when performing VDI and AES operations on the same rectangle.

Because the AES library calls frequently require a rectangle description, some programs make use of a data structure that includes all of the information required for such a description. This structure is defined in the header file OBDEFS.H, which is included with most C compilers for the ST, as follows:

```
typedef struct grect
{
    int g_x;
    int g_y;
    int g_w;
    int g_h;
} GRECT;
```

The use of *typedef* means that you can declare a structure of the type grect either by using the declaration form

```
struct grect rectangle;
```

or the form

```
GRECT rectangle;
```

In either case, the *x* coordinate is denoted by rectangle.g_x, the *y* coordinate by rectangle.g_y, the width by rectangle.g_w, and the height by rectangle.g_h. Using such a data structure makes it possible to reference all the necessary information about a rectangle using a single variable or pointer.

As the function template above shows, the wind_create( ) function returns a value called *wi_handle*. If the function successfully allocates a window, the value returned is a unique ID number, known as the *window handle*. This window handle is used to identify the window for purposes of the Window Library routines, which can modify a window or return information about it. A handle of 0 is reserved for the GEM

Desktop window. A value in the range 1–8 is used for the application windows.

If the AES is unable to allocate a new window, a negative value is returned in wi_handle. In the current version of GEM on the ST, the AES will only let you create up to eight application windows. This number includes the windows that are opened by desk accessories. If you want your program to work with desk accessories, leave some of the available windows for them. The GEM Desktop program limits itself to four windows, so that the other four can be used by desk accessories. If a negative handle is returned by wind_create( ), you may wish to display an alert telling the user to close a window and try again, if possible.

As stated above, most windows are created with a maximum size that matches that of the Desktop window's work area. One way of finding out the dimensions of that rectangle is to use the *wind_get( )* function. This function can be used to return any of several items of information concerning a window. Its syntax is

```
int status, wi_handle, flag, x, y, width, height;
status = wind_get(wi_handle, flag, &x, &y, &width, &height);
```

where *wi_handle,* is the window handle returned by wind_create( ), which identifies the window. The next input parameter, *flag,* determines what kind of information is returned about the window. The following table shows the valid values for flag and the information returned by the call when each flag is used:

Flag	Macro Name	Information Requested
4	WF_WORKXYWH	Window work area coordinates
5	WF_CURRXYWH	Window exterior coordinates
6	WF_PREVXYWH	Previous window exterior coordinates
7	WF_FULLXYWH	Maximum window exterior coordinates
8	WF_HSLIDE	x = relative position of horizontal slider (1 = leftmost position, 1000 = rightmost)
9	WF_VSLIDE	x = relative position of vertical slider (1 = top position, 1000 = bottom)
10	WF_TOP	x = window handle of the top (active) window
11	WF_FIRSTXYWH	Coordinates of the first rectangle in the window's rectangle list

Flag	Macro Name	Information Requested
12	WF_NEXTXYWH	Coordinates of the next rectangle in the window's rectangle list
13	WF_RESVD	Reserved for future use
15	WF_HSLSIZE	x = relative size of the horizontal slider compared to the scroll bar (-1 = minimum size, 1–1000 = percentage filled, in tenths of 1 percent)
16	WF_VSLSIZE	x = relative size of the vertical slider compared to the scroll bar (-1 = minimum size, 1–1000 = percentage filled, in tenths of 1 percent)
17	WF_SCREEN	Address and length of the menu/alert buffers (x = low word of address, y = high word, *width* = low word of length, *height* = high word)

Each of the values for flag has a C macro name that's related to its function. These macros are defined in the header file GEMDEFS.H which is included with most C compilers. The flags under consideration here are the ones that return the window size. These include WF_WORKXYWH, WF_CURRXYWH, WF_FULLXYWH, and WF_PREVXYWH. When any of these flags except WF_WORKXYWH is used, the function returns the size of the exterior outline of the window. This includes the border and any window controls located in the border, such as the title bar or scroll bars. When the WF_WORKXYWH flag is used, the function returns the size of the interior area of the window only. If you restrict your drawing to that rectangle, you'll never draw over a window border control box by mistake.

The other flag values, which deal with such information as slider size and position and the rectangle list, will be covered in subsequent chapters. Note however, that some of these functions return values other than the standard rectangle *x, y, width* and *height*. In most cases, a single value is returned in place of the *x* coordinate.

As stated above, a window handle of 0 is reserved for the Desktop window. Therefore, to find the dimensions of that window, which represents the maximum free area available for the applications window, you can use the call

```
#include <GEMDEFS.H>
int status,deskx, desky, deskw, deskh;
status = wind_get(0,WF_WORKXYWH, &deskx, &desky,
&deskw, &deskh);
```

The Desktop window dimensions (excluding the menu bar) will be returned in *deskx, desky, deskw,* and *deskh.* These values can then be used as the input parameters for the maximum window size in wind_create( ). The status value indicates whether or not there's been an error. If status is 0, an error has occurred, and if it's greater than 0, there's no error. Note also that if you make this call using the subcommand WF_CURRXYWH instead of WF_WORKXYWH, you get the entire screen size, including the menu bar.

Opening a Window

Just creating a window doesn't cause that window to be displayed on screen. For that, you must open the window using *wind_open( ).* If, however, you specified in the wind_create( ) function that the window controls attached to this window should include either a title bar or information line, there's one additional step that you must take before opening the window: You must tell the window where to find the text for the information line or window title. The call to use for this purpose is *wind_set( ).* The syntax for this call is

```
int status, wi_handle, field, x, y, width, height;
status = wind_set(wi_handle, field, x, y, width, height);
```

Just as the wind_get( ) call retrieves several different bits of information about the window, wind_set( ) allows you to change various aspects of the window's appearance. Select the aspect you wish to change with the field parameter. This parameter may have any of the following values:

Field Number	Name	Aspect to Change
1	WF_KIND	*x* = Window controls flag (same as controls for wind_create( ))
2	WF_NAME	*x,y* = Address of string containing the name of the window
3	WF_INFO	*x,y* = Address of string for the window's information line
5	WF_CURRXYWH	Window exterior coordinates

Field Number	Name	Aspect to Change
8	WF_HSLIDE	*x* = relative position of horizontal slider (1 = leftmost position, 1000=rightmost)
9	WF_VSLIDE	*x* = relative position of vertical slider (1 = top position, 1000 = bottom)
10	WF_TOP	*x* = window handle of the top (active) window
14	WF_NEWDESK	The address of an object tree to be used for the Desktop Window background (*x* = low word, *y* = high word of address, *width* = number of starting object to draw)
15	WF_HSLSIZE	*x* = relative size of the horizontal slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of 1 percent)
16	WF_VSLSIZE	*x* = relative size of the vertical slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of 1 percent)

Again, the macro names are synonymous with the field numbers defined in the header file GEMDEFS.H. A status value of 0 indicates that an error has occurred, while a value greater than 0 means no error.

As with wind_get( ), some of the field types use fewer input parameters. The two fields of interest here, WF_NAME and WF_INFO require only a pointer to a text string. Since the pointer is a four-byte long word, it takes the place of both of the integer input values *x* and *y*. The format used by both of these calls is

```
#include <GEMDEFS.H>
static char *string = "Window Title";
int wi_handle;

wind_set(wi_handle, WF_NAME, string, 0, 0);
```

Or the following could be used since, in C, the string "Window Title" is treated as a pointer to a static array of the type char:

```
wind_set(wi_handle, WF_NAME, "Window Title", 0, 0);
```

Note that you must use a static array, since the AES will periodically look at this title in order to redraw the title bar when the window is sized. If the array is not permanent, the pointer to the string might be rendered invalid. In such a case, the AES might try to access an invalid string, with disastrous results.

Once you've set the pointers to the window title and information line strings (if necessary), you're ready to open the window. The format for the *wind_open( )* call used to perform this function is

```
int status, wi_handle, x, y, width, height;
status = wind_open(wi_handle, x, y, width, height);
```

where *x, y, width,* and *height* describe the initial exterior dimensions of the window. These dimensions may be smaller than the full-size window described in wind_create( ), but they may not exceed those dimensions. The *status* value returns the error status of the function. A 0 indicates that an error has occurred, while a value greater than 0 means no error.

As with wind_create( ), the dimensions used to describe the window in wind_open( ) measure the exterior of the window and include the borders and any controls located within those borders. Your program, however, must confine its output to the interior area of the window, so as to avoid overwriting the window borders. You can use the wind_get( ) function to find the window's interior dimensions as follows:

```
#include <GEMDEFS.H>
int status,wi_handle,workx, worky, workw, workh;
status = wind_get(wi_handle,WF_WORKXYWH, &workx,
&worky, &workw, &workh);
```

While wind_get( ) can be used to find the current interior or exterior dimensions of a window, it isn't of much help before you've created the window. Let's say, for example, that you want to open a window whose interior work area will be 200 × 100 pixels and whose top left corner will be located at 50,50. How can you determine the exterior dimensions of such a window, in order to pass them to the wind_open( ) call? One way is to use the *wind_calc( )* function. Given the window control flag and either the interior or exterior dimensions of the window, this function can return the opposite set of dimensions. Its syntax is

```
int status, type, controls, knownx, knowny, knownw, knownh,
   otherx, othery, otherw, otherh;
status = wind_calc (type, controls, knownx, knowy, knownw,
   knownh, &otherx, &othery, &otherw, &otherh);
```

The dimensions of the known window area are specified by *knownx, knowny, knownw,* and *knownh.* The function returns the other set of dimensions in *otherx, othery, otherw,* and *otherh. Type* is a flag byte showing what type of conversion to perform. A 0 value means to consider the known dimensions to be the interior area, so that the function returns the exterior dimensions. A value of 1 causes the function to return the interior dimensions of the window.

The *controls* input parameter is the same type of flag used by wind_create( ) to indicate the types of controls to attach to the window. These are significant to calculating the size of the window because their presence may enlarge the border area. If a window has a close box, title bar, move bar, or full box, this extends the border area at the top of the box to the height of a character box of the default character set. This is the value returned in the chboxh variable by the graf_handle( ) call. If the window has an information line, the top border is enlarged by the height of an additional character cell. If there is a vertical slider, or up or down arrows, the right border is extended to the width of the default character cell. This value is returned in the chboxw variable by the graf_handle( ) call. Finally, if there's a size box, horizontal slider, or right or left arrows, the bottom border is extended to the height of the default character cell.

Closing a Window

When you're through using a window for displaying output, you may close it using *wind_close( )*. The format for this call is

```
int wi_handle, status;
status = wind_close(wi_handle);
```

where *status* contains the error status of the call (0 means there was an error, while a value greater than 0 indicates no error). Wind_close( ) removes the window's display from the screen by sending a message to the other windows on the screen to update their display. If the window wasn't covering any application windows, its image is replaced by the Desktop background. While wind_close( ) removes the window display from the screen, the window remains allocated, and may be reopened at any time by a call to wind_open( ). In order to remove the window completely, you must call *wind_delete( )* as follows:

```
int wi_handle, status;
status = wind_delete(wi_handle);
```

Once you've deleted a window, you must use wind_create( ) to reallocate it before you can open it again. You should always remember to close and delete all of your windows before your program terminates. Closing them will remove their image from the screen, and deleting them will return the resources they use to the system.

A C Program Shell

Since most of the subsequent example programs use much of the same program code for initialization and cleanup, it would be repetitive to include the text of that code in every example. Instead, the steps necessary to open a virtual workstation and a window are listed below, in the form of a short program named *aesshell.c.* All this program does is perform AES and VDI initialization, open a window, call a function named *demo,* close the window and exit.

A few things are missing from this shell, however, so the program will not link properly or run unless you add them yourself. You have to add the function named demo, and macro definitions for the window characteristics WDW_CTRLS, APP_NAME, and APP_INFO. The example will accomplish this through the use of the C #include directive to include the file aesshell.c at the beginning of most of the sample programs. The main function of the sample program will be called demo( ), and there will be macro definitions for the window characteristics. For example, to create a program that does absolutely nothing but open a window and wait for the user to click on its close box, you could use the code in Program 2-1.

Program 2-1. dummy.c

```
/**********************************************************************/
/*                                                                    */
/*      DUMMY.C -- A short calling program that uses the              */
/*      program template  AESSHELL.C.  It defines the                 */
/*      window name and the window controls to be included.           */
/*      It then waits a while, and exits.                             */
/*                                                                    */
/*                                                                    */
/**********************************************************************/

#define APP_INFO " ** Click on the Close Box to exit the program. **"
#define APP_NAME "Dummy Title"
#define WDW_CTRLS (NAME|CLOSER|INFO)

#include "aesshell.c"

demo()
{
  int msgbuf[8];

    evnt_mesage(&msgbuf);   /* skip the window redraw message, */
    evnt_mesage(&msgbuf);   /* & wait for the window close message */

}
```

Keeping the initialization code in a reusable file will shorten the sample listings substantially, and eliminate retyping. Be sure the file aesshell.c is stored where your compiler can find it, either in the same disk and directory as your standard header files or in the same disk and directory as the source-code file. Program 2-2 is the text of the aesshell.c program shell.

Program 2-2. aesshell.c

```
/**********************************************************************/
/*                                                                    */
/*      AESSHELL.C -- Program template to be included with            */
/*      most of the AES example programs.  Performs init-             */
/*      ialization functions, calls the demo program,                 */
/*      then does the cleanup work.                                   */
/*                                                                    */
/**********************************************************************/

#include <osbind.h>     /* Macro definitions for BIOS calls */
#include <gemdefs.h>    /* Flag definitions for Library routines */
#include <obdefs.h>     /* Object definitions */

#define DESK 0          /* The flag for the Desktop Window */
#define NO_ERR 0        /* Error no. for "no error" */
#define APP_ERR 1       /* Error no. for failure of appl_init() */
#define VWK_ERR 2       /* Error no. for failure of v_opnvwk() */
#define WDW_ERR 3       /* Error no. for failure of wi_create() */
```

```
/* Global variables -- For VDI bindings and program routines */

extern int gl_apid;    /* The application ID part of the global array */
int ap_id;

int contrl[12],        /* VDI data arrays */
    intin[128],
    ptsin[128],
    intout[128],
    ptsout[128];

int phys_handle,     /* workstation handle for physical screen device */
     handle,         /* workstation handle for virtual screen device */
     wi_handle;      /* window handle */

int work_in[12],     /* input and output arrays for v_opnvwk() */
    work_out[57];

GRECT desk, work;    /* Desktop and application window dimensions */


int cellw, cellh, chspcw, chspch;  /* size of default character font */


/* Program starts here */

main()
{
    int error;              /* Error flag */

    error = init_all();     /* Initialize application, open workstation, */
                            /* and open application window */
    if (!error) demo();     /* If no initialization failures, run demo */
    cleanup(error);         /* Close window, workstation, and application */

 }

/* Initialize application, open graphics workstation, and open window */

 init_all()
 {
    int x;
    int points[4];

 /* Initialize the GEM application.  If this fails, return error code. */

    appl_init();
    ap_id=gl_apid;
    if (ap_id==-1) return (APP_ERR);

/* Initialize input array, get the physical workstation handle,
   and open the Virtual Screen Workstation for VDI calls. */

    handle = phys_handle = graf_handle(&cellw, &cellh, &chspcw, &chspch);
                              /* get physical screen device handle */
    work_in[10]=2;            /* use Raster Coordinates */
    work_in[0]=Getrez()+2;    /* set screen device ID according to */
                              /*  resolution mode.  */
    for (x=1; x<10; work_in[x++]=1);
                              /* set other input values to default */
    v_opnvwk (work_in, &handle, work_out);
                              /* open virtual screen workstation */
    if (handle == 0)return(VWK_ERR);
                              /* if we can't open it, return error code */
```

```
/*  Find out the maximum size for a window, and open one.  */

    wind_get(DESK, WF_WORKXYWH, &desk.g_x, &desk.g_y,
            &desk.g_w, &desk.g_h);
                            /* find dimensions of Desktop Window */
    wi_handle = wind_create(WDW_CTRLS, desk.g_x, desk.g_y,
                            desk.g_w, desk.g_h);
                            /* Create a window that size */
    if (wi_handle<0) return(WDW_ERR);
                            /* if we can't, return error code */
    wind_set(wi_handle,WF_INFO, APP_INFO,0,0);
    wind_set(wi_handle,WF_NAME, APP_NAME,0,0);
                            /* set name and info string for window */
    wind_open(wi_handle, desk.g_x, desk.g_y, desk.g_w, desk.g_h);
                            /* open the window to full size */


/* Clear the work area of the window */


    wind_get(wi_handle, WF_WORKXYWH, &work.g_x, &work.g_y,
            &work.g_w, &work.g_h);
                            /* find out the size of its work area */

    graf_mouse(M_OFF, 0L);      /* turn the mouse pointer off */
    clear_rect(&work);          /* clear the area */
    graf_mouse(ARROW, 0L);      /* change the pointer to an arrow */
    graf_mouse(M_ON, 0L);       /* and turn it back on */

    return(0);                  /* Report no errors */
 }


/* Close and delete window, close workstation, exit application */


 cleanup(error)
    int error;

{
     switch(error)        /* Perform as much cleanup as is warranted  */
                          /* by the error level.  Each higher level   */
                          /* falls through to subsequent lower levels */
     {
     case NO_ERR:         /* if no error, close window */
        wind_close(wi_handle);
        wind_delete(wi_handle);
     case WDW_ERR:        /* If couldn't create window, close workstation */
        v_clsvwk(handle);
     case VWK_ERR:       /* If workstation didn't open, exit app */
        appl_exit();
     case APP_ERR:       /* if appl_init() failed, exit immediately */
        ;
     }
}

/* >>>>>>>>>>>>>>>>>>> Some Handy Functions <<<<<<<<<<<<<<<<< */

clear_rect(r)  /* clear a rectangle to the background color */
   GRECT *r;
{
  int points[4];

  vsf_interior(handle,0);
  grect_conv(r, &points);
  vr_recfl(handle, points);
}
```

```
grect_conv(r, array)    /* convert grect to an array of points  */
   GRECT   *r;
   int  *array;
   {
   *array++ = r->g_x;
   *array++ = r->g_y;
   *array++ = r->g_x + r->g_w - 1;
   *array = r->g_y + r->g_h - 1;
   }
```

This program is divided into three parts. The first, *init_all( )*, registers the GEM application, opens a Virtual Screen Workstation for possible VDI calls, and then creates and opens a window. It uses the VDI function *vr_recfl( )*, discussed in Volume 1 of this series, to clear the work area of the window. And it uses the *graf_mouse( )* function to turn off the mouse pointer and change its shape (the graff_mouse function will be discussed in greater detail later).

If the essential initialization steps are not completed successfully, init_all( ) returns the appropriate error code. If no error occurred, the user-supplied *demo( )* routine is executed. Finally, the *cleanup( )* module closes and deletes the window, closes the workstation, and exits the program.

A Machine Language Program Shell

Setting up a bare-bones machine language program is more involved than just translating the corresponding shell.c program. For one thing, C programs usually link in a startup file at the beginning of the program to take care of such maintenance chores as allocating RAM for a program stack, setting the stack pointer to the address of that that stack, and returning any unused RAM to the pool of free memory. Programs written with *Alcyon* C link in the file appstart.o or gemstart.o at the beginning to take care of these tasks, and *Megamax* C programs get the necessary code from a library module called init.o, the source code for which is supplied in a file called init.c (it uses the inline assembly commands). But machine language programmers must provide the equivalent functions for each of their programs themselves.

The other problem is that not all assemblers have an include directive, so you won't be able to include the text of the shell program in each of our demo programs. Instead, assemble the shell program separately and link the resulting object file with the demo program object files.

Since the shell program refers to the demo subroutine in the demo program file, and the demo programs refer to the VDI data arrays defined in the shell program, use the .xdef and .xref directives to help resolve these external references. The .xref directive tells the assembler that the symbol is defined in another object file, while .xdef tells it that this symbol will be used by another object file.

All of the machine language examples in this book have been created to be assembled with the *Megamax* C compiler. If you have the *Alcyon* compiler, use this short batch file to assemble the machine language programs with *Alcyon*.

```
as68 -l -u %1.s
link68 [u] %1.68k=%1
relmod %1
rm %1.68k
rm %1.o
wait %1
```

Type in this file with your editor and save it to disk with the filename AES.BAT.

Next, call the batch environment by double-clicking on the *Alcyon* system file BATCH.TTP. For parameters, type AES *filename*, where *filename* is the name of the source file you want to assemble.

Program 2-3 is the assembler shell program, aesshell.s.

Program 2-3. aesshell.s

```
****************************************************************
*                                                              *
*  AESSHELL.S  -- Shell program to be linked with all          *
*  assembly language example programs.                         *
*                                                              *
****************************************************************

*** Program equates

bpadr    =     4   * Stack offset to base page address
codelen  =    12   * Base page offset to Code segment length
datalen  =    20   * Base page offset to Data segment length
bsslen   =    28   * Base page offset to BSS  segment length
stk      =  $400   * size of our stack (1K)
bp       =  $100   * size of base page

setblk   =  $4a    * command number of SETBLOCK function
aescode  =  $c8    * command number for AES call
vdicode  =  $73    * command number for VDI call

*** External references

** Import:

.xref    demo      * the external demo subroutine.
```

```
.xref    wdwctrl
.xref    wdwtitl
.xref    wdwinfo

** Export:

.xdef    aes
.xdef    vdi

.xdef    pwkhnd    * the phyusical workstation handle,
.xdef    vwkhnd    * virtual workstation handle,
.xdef    wdwhnd    * window handle,
.xdef    contrl0   * all of the VDI data arrays
.xdef    contrl1
.xdef    contrl2
.xdef    contrl3
.xdef    contrl4
.xdef    contrl5
.xdef    contrl6
.xdef    contrl7
.xdef    contrl8
.xdef    contrl9
.xdef    contrl10
.xdef    contrl11
.xdef    intin
.xdef    intout
.xdef    ptsin
.xdef    ptsout
.xdef    ctrl0   * all of the AES data arrays
.xdef    ctrl1
.xdef    ctrl2
.xdef    ctrl3
.xdef    ctrl4
.xdef    aintin
.xdef    aintout
.xdef    addrin
.xdef    addrout
.xdef    global
.xdef    apid

.xdef    chboxw      * and miscellaneous work variables
.xdef    chboxh
.xdef    cellw
.xdef    cellh
.xdef    deskx
.xdef    desky
.xdef    deskw
.xdef    deskh
.xdef    workx
.xdef    worky
.xdef    workw
.xdef    workh


*** Program starts here.  Get base page address in a5

   .text
   move.l   a7,a5          * save a7 so we can get the base  page address
   move.l   bpadr(a5),a5   * a5 = basepage address

*** Calculate the total amount of memory used by
*** our program (including stack space) in d0

*                           * total memory used =
   move.l   codelen(a5),d0  *   length of code segment
   add.l    datalen(a5),d0  * + length of data segment
   add.l    bsslen(a5),d0   * + length of uninitialized storage segment
   add.l    #stk+bp,d0      * + (size of the base page + our stack)
```

```
*** Calculate the address of our stack
*** and move it to the stack pointer (a7)

*                         * stack address =
   move.l   dØ,d1         *   size of program memory
   add.l    a5,d1         * + program's base address,
   and.l    #-2,d1        * pick off odd bit to make sure that the
*                         * stack starts on a word boundary (it must).
   move.l   d1,a7         * set stack pointer to our stack
*                         * which is stk bytes above end of BSS

*** Use the GEMDOS SETBLOCK call to reserve the area of memory
*** actually used for the program and stack, and release the
*** rest back to the free memory pool.

   move.l dØ,-(sp)        * push the size of program memory
*                         * (first SETBLOCK parameter) on the stack.
   move.l a5,-(sp)        * push the beginning address of the
*                         * program memory area (2nd SETBLOCK parameter).
   clr.w   -(sp)          * clear a dummy place-holder word
   move    #$4a,-(sp)     * finally, push the GEMDOS command number
*                         * for the SETBLOCK function
   trap    #1             * call GEMDOS
   add.l   #12,sp         * and clear our arguments off the stack.

*** Initialize the application with appl_init

   move.l    #Ø,resv1         * clear global variables
   move.l    #Ø,resv2
   move.l    #Ø,resv3
   move.l    #Ø,resv4
   move      #10,ctrlØ    * command = appl_init
   move      #Ø,ctrl1     * no integer input parameters
   move      #1,ctrl2     * 1 integer output parameter
   move      #Ø,ctrl3     * no address input parameters
   move      #Ø,ctrl4     * no address output parameters
   jsr       aes              * do the call

   cmpi      #$FFFF,apid  * check to see if init failed
   beq       apperr       * and exit if it did

*** Get the physical screen device handle from graf_handle

   move      #77,ctrlØ    * command = graf_handle
   move      #5,ctrl2     * 5 integer output parameters
   jsr       aes              * do the call
   move      aintout,pwkhnd   * save handle and char sizes
   move      aintout+2,cellw
   move      aintout+4,cellh
   move      aintout+6,chboxw
   move      aintout+8,chboxh

*** Open the Virtual Screen Workstation (v_opnvwk)

   move      #1ØØ,contrlØ   * opcode to contrl(Ø)
   move      #Ø,contrl1     * no points in ptsin
   move      #11,contrl3    * 11 integers in intin
   move      aintout,contrl6 * physical workstation handle to contrl(6)

   movea.l   #intin+2,aØ    * destination address
   move      #8,dØ          * loop counter
initloop:
   move.w    #1,(aØ)+       * intin(1)-intin(9) = 1
   dbra      dØ,initloop
```

```
    move      #2,intin+20      * intin(10) = 2 (Raster Coordinates)
    move.w    #4,-(sp)         * push getrez command on stack
    trap      #14              * call XBIOS
    addq.l    #2,sp            * pop command off stack
    addq      #2,d0
    move      d0,intin         * use rez+2 as device ID

    jsr       vdi
    move      contrl6,vwkhnd * save virtual workstation handle
    beq       vwkerr           * end program if it's zero

*** Find max window size

    move      #104,ctrl0   * command = wind_get
    move      #2,ctrl1     * 2 input integers
    move      #0,aintin        * window handle of Desktop
    move      #4,aintin+2      * WF_WORKXYWH command

    jsr       aes
    move      aintout+2,deskx  * store desk x,y,w,h
    move      aintout+4,desky
    move      aintout+6,deskw
    move      aintout+8,deskh

***  Create a window with that max size

    move      #100,ctrl0   * command = wind_create
    move      #5,ctrl1     * 5 input integers
    move      #1,ctrl2     * 1 ouput integer
    move      wdwctrl,aintin * window ctrl flag
    move      deskx,aintin+2 * max x
    move      desky,aintin+4 * max y
    move      deskw,aintin+6 * max width
    move      deskh,aintin+8 * max height

    jsr       aes
    move      aintout,wdwhnd * save window handle
    bmi       wdwerr           * if negative, exit program

*** set window name

    move      #105,ctrl0   * command = wind_set
    move      #6,ctrl1     * 6 input integers
    move      wdwhnd,aintin  * window handle
    move      #2,aintin+2     * subcommand = set window name
    move.l    #wdwtitl,aintin+4 * point to title

    jsr       aes

*** set info line

    move      #3,aintin+2     * subcommand = set info line
    move.l    #wdwinfo,aintin+4 * point to info text

    jsr       aes

*** Open the window

    move      #101,ctrl0   * command = wind_open
    move      #5,ctrl1
    move      #1,ctrl2     * 1 ouput integers
    move      deskx,aintin+2 * initial x
    move      desky,aintin+4 * initial y
    move      deskw,aintin+6 * initial width
    move      deskh,aintin+8 * initial height

    jsr       aes
```

```
*** Find window work area size

   move      #104,ctrl0   * command = wind_get
   move      #2,ctrl1     * 2 input integers
   move      #5,ctrl2     * 5 ouput integers
   move      #4,aintin+2    * WF_WORKXYWH command

   jsr       aes
   move      aintout+2,workx  * store work x,y,w,h
   move      aintout+4,worky
   move      aintout+6,workw
   move      aintout+8,workh

*** set fill color to white

   move    #25,contrl0    * opcode for set fill color (vsf_color)
   move    #0,contrl2
   move    #1,contrl3
   move    #1,contrl4
   move    #0,intin       * select white

   jsr       vdi

*** turn mouse off

   move      #78,ctrl0    * command = graf_mouse
   move      #1,ctrl1     * 1 input integers
   move      #1,ctrl2     * 1 output integer
   move      #256,aintin * hide the mouse

   jsr       aes

*** fill work area of window with white

   move    #114,contrl0    * opcode for fill rectangle (vr_recfl)
   move    #2,contrl1
   move    #0,contrl3
   move    #0,contrl4

   move    workx,d0
   move    d0,ptsin
   add     workw,d0
   subq    #1,d0
   move    d0,ptsin+4
   move    worky,d0
   move    d0,ptsin+2
   add     workh,d0
   subq    #1,d0
   move    d0,ptsin+6

   jsr       vdi

*** change mouse to arrow

   move      #0,aintin    * set mouse to arrow shape

   jsr       aes

*** turn mouse back on

   move      #257,aintin * show the mouse

   jsr       aes

*** Do our demo program

  jsr demo
```

```
*** Close the Window
   move     #102,ctrl0  * command = wind_close
   move     #1,ctrl1
   move     #1,ctrl2
   move     #0,ctrl3
   move     #0,ctrl4
   move     wdwhnd,aintin

   jsr      aes

*** Delete the Window

   move     #103,ctrl0  * command = wind_delete

   jsr      aes

*** Close Virtual Screen Workstation  (v_clsvwk)
wdwerr:
   move     #101,contrl0    * opcode to contrl(0)
   move     #0,contrl1      * no points in ptsin
   move     #0,contrl3      * no integers in intin

   jsr      vdi

*** Finish the application (appl_exit)
vwkerr:
   move     #19,ctrl0     * opcode to contrl(0)
   move     #0,ctrl1

   jsr      aes

*** Exit back to DOS
apperr:
   move.l #0,(a7)      * Push command number for terminate program
   trap   #1           * call GEMDOS.  Bye bye!


***   Make AES function call
***   (after setting parameters)

aes:
   move.l   #apb,d1
   move.w   #aescode,d0
   trap   #2
   rts

*** Make VDI function call
*** (after setting parameters)

vdi:
   move.l   #vpb,d1
   move.w   #vdicode,d0
   trap   #2
   rts


*** Storage space for AES and VDI call parameters

.data
.even
```

CHAPTER 2

```
********** VDI Data Arrays ************

contrl:
contrl0:     .ds.w 1
contrl1      .ds.w 1
contrl2:     .ds.w 1
contrl3:     .ds.w 1
contrl4:     .ds.w 1
contrl5:     .ds.w 1
contrl6:     .ds.w 1
contrl7:     .ds.w 1
contrl8:     .ds.w 1
contrl9:     .ds.w 1
contrl10:    .ds.w 1
contrl11:    .ds.w 1

intin:       .ds.w 128
intout:      .ds.w 128
ptsin:       .ds.w 128
ptsout:      .ds.w 128


************** AES Data Arrays ****************
ctrl:
ctrl0:    .ds.w 1
ctrl1     .ds.w 1
ctrl2:    .ds.w 1
ctrl3:    .ds.w 1
ctrl4:    .ds.w 1

global:
version:     .ds.w 1
count:       .ds.w 1
apid:        .ds.w 1
private:     .ds.l 1
tree:        .ds.l 1
resvl:       .ds.l 1
resv2:       .ds.l 1
resv3:       .ds.l 1
resv4:       .ds.l 1

aintout:     .ds.w 8
aintin:      .ds.w 18
addrin:      .ds.l 3
addrout:     .ds.l 2


*** The AES and VDI parameter blocks hold pointers
*** to the starting address of each of the data arrays

apb:  .dc.l ctrl,global,aintin,aintout,addrin,addrout
vpb:  .dc.l contrl,intin,ptsin,intout,ptsout


****** Misc variables *********

vwkhnd    .ds.w 1
pwkhnd    .ds.w 1
wdwhnd    .ds.w 1

chboxw    .ds.w 1
chboxh    .ds.w 1
cellw     .ds.w 1
cellh     .ds.w 1
```

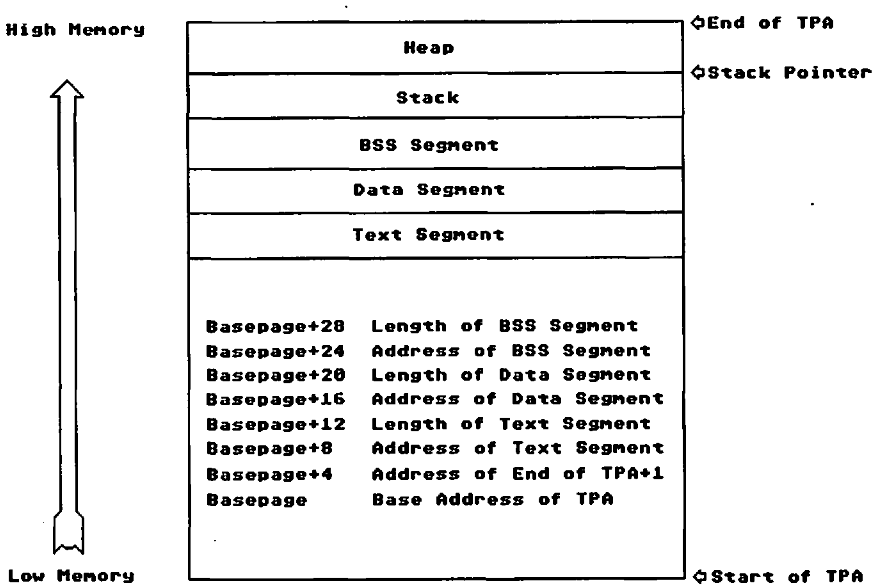
```
deskx    .ds.w 1
desky    .ds.w 1
deskw    .ds.w 1
deskh    .ds.w 1

workx    .ds.w 1
worky    .ds.w 1
workw    .ds.w 1
workh    .ds.w 1

.end


        .xdef     demo
        .xdef     wdwctrl
        .xdef     wdwtitl
        .xdef     wdwinfo


        .xref     aes
```

The first part of the program requires a bit of explanation. When GEM starts an application program (but not a desk accessory), it allocates all of the system memory to that program. Therefore, if the program wishes to use the system memory-management calls, or any of the AES calls that themselves allocate memory, it must deallocate all of the memory it isn't actually using, at startup time. The way to do this is with the XBIOS function, SETBLOCK. SETBLOCK is used to reserve a specific area of memory for the program and return the remaining RAM area to the Operating System's free memory pool. In order to execute this command, you must pass the starting address of the area you wish to reserve and the size of the area. Please remember that it's only necessary to free memory when you start an application program. It's not necessary to do so with a desk accessory.

Finding the starting address of program memory isn't difficult. When you start the program, the second word on the stack points to that location. Finding the size of the program requires a little more knowledge of how program storage space is allocated.

The memory area in which a program resides is known as the Transient Program Area (TPA). At the beginning of the TPA is a 256-byte segment known as the *basepage*. The basepage contains information about the size and address of each program segment, as well as the command line that is passed to the program (these are the extra characters you type in when you run a TOS Takes Parameters program whose name ends in .TTP). After the basepage comes the actual program code, followed by the data area and the BSS (Block Storage Segment), which is used to store uninitialized data. To find the total size of the program area, look in the basepage area to find the size of the code. Add that to the size of the data and BSS segments, along with the size of the basepage itself. Since you need a stack area for the program, it makes sense to add the size of the stack to the end of the program and reserve the combined program and stack area together (Figure 2-2). Once you calculate this area, you can set the stack pointer to the top of program memory and make the SETBLOCK call. When that is done, continue on with whatever your program does.

Figure 2-2. Layout of Transient Program Area in ST Memory



In order to assemble the aesshell.s program with the *Alcyon* assembler, invoke the as68 assembler with the following command:

as68 -u -l aesshell.s

This creates an object file called aesshell.o. Since this program does not contain the demo subroutine or the window-definition constants, it won't link and run properly. In order to get it to function, you must create another object module that contains that subroutine. An example of this is Program 2-4, dummy.s.

Program 2-4. dummy.s

```
************************************************************
*                                                          *
*     DUMMY.S Just waits for user to click close box       *
*                                                          *
************************************************************

*** External References

** Export:


     .xdef     demo
     .xdef     wdwctrl
     .xdef     wdwtitl
     .xdef     wdwinfo

** Import:

     .xref     aes

     .xref     ctrl0
     .xref     ctrl1
     .xref     ctrl2
     .xref     ctrl3
     .xref     ctrl4
     .xref     addrin

       .text

     demo:
       jsr demo1
       jsr demo1


     demo1:
        move      #23,ctrl0
        move      #0,ctrl1
        move      #1,ctrl2
        move      #1,ctrl3
        move      #0,ctrl4
        move.l    #msgbuf,addrin
        jmp       aes
```

```
*** Storage space and data constants

     .data
     .even

     msgbuf:    .ds.w 8

     wdwtitl:   .dc.b 'Test Window',0
     wdwinfo:   .dc.b 'Info Line',0
     wdwctrl:   .dc.w 55


       .end
```

Assemble this file in the same way to create the dummy.o file. Next, use the linker to join the two object modules. The command line to use is

link68 [u] dummy.68k=aesshell,dummy

This creates the dummy.68k file, a program module that must be modified to run under GEMDOS, using the relmod program:

relmod dummy

This produces the dummy.prg program file that can be executed from the desktop. You may have noticed that the source code for the assembler shell program is about twice as long as that of the C shell program. But depending on what compiler you use, the executable program generated by the assembler version is at least 50-percent smaller than the C program. In order to reduce the size of the assembler source code, a number of shortcuts were used. Since you know the contents of the AES data arrays like *ctrl* (the name given to the control array) at any given time, you don't need to fill each member for each function call. For example, since the input parameters for the *graf_mouse( )* call used to change the mouse form to an arrow are almost exactly the same as the ones used for the graf_mouse( ) call that turns the mouse back on, the only input variable that is changed between the two calls is *aintin* (the name given to the first element of the *int_in* array).

Chapter 3

AES Events: Windows, Part 2

In older, single-tasking microcomputer systems, it's common for a program to check for input from the user by polling the input devices. That means the program sits in a loop, continuously checking the device until something happens. It's sort of like having a telephone without a bell—you have to pick it up every few seconds in order not to miss any calls. In a single-tasking system, this kind of programming is appropriate, since the processor literally doesn't have anything else to do. But in a multitasking system—even a limited multitasking system such as GEM—such programming techniques aren't adequate. While one task sits and waits, all of the other tasks are slowed down or shut out completely. Fortunately, GEM provides a much better method of waiting for input, known as event-oriented waiting.

Using the AES Event Library routines, a program tells the AES that it must wait until a specified event happens. The multitasking kernel then puts the program on the Not-Ready list and lets the other tasks that are on the Ready list share the processor's time. After the specified event occurs the kernel puts the program back on the Ready list and lets it execute again.

The AES Event Library allows the program to wait for a number of different types of I/O events. These include the usual types of direct input from the user, such as typing on the keyboard, moving the mouse, and clicking a mouse button. These I/O events also include a more sophisticated, indirect type of input, called a *message*. The Screen Manager sends these messages to let the program know that the user has performed a significant action, such as clicking on a window's close box or full box, or selecting an item from a drop-down menu. Messages may also be generated by system events, such as the window-redraw messages that are sent when a window's graphics have been damaged by moving or sizing other windows. The system timer may be used to generate an event after a specified length of time.

Since most programs need to check for more than one type of event, a function called *evnt_multi( )* is provided, which may be used to wait for any combination of events. In a typical GEM program, the main program loop centers around an evnt_multi( ) call, and the various routines that are used to handle the events returned by this call.

Message Events

In order for a multitasking system such as GEM to be really effective, there has to be a way for tasks to communicate with one another, so that there's an orderly division of labor. GEM provides a message system for intertask communication. Each task has its own message pipe, in which messages are stored in FIFO (First In, First Out) order. When a task asks for its messages, the top one is taken from the pipe and moved to a buffer which the task designates. The message pipe can hold up to eight 16-byte messages at a time.

When an applications wants to wait for a message, it uses the *evnt_mesag( )* call, whose format is as follows:

```
int reserved, msgbuf[8];
reserved = evnt_mesag(msgbuf);
```

where the *msgbuf* array is a temporary storage buffer in which the 16-byte message is deposited. The reserved variable is always equal to 1. The fact that a reserved variable is provided indicates that this function may return some significant value in the future.

If messages are already waiting in the message pipe, the evnt_mesag( ) call will return immediately with the first one. If there are no messages in your program's message pipe, a call to evnt_mesag( ) will force your program to wait until one is received.

The values actually placed in the msgbuf array depend on the type of message sent. The format for the first three words of each message is standardized:

Element Number	Contents
0	Message ID (indicates type of message)
1	Application ID of message sender
2	Number of additional bytes in message (in excess of the standard 16)
3–7	Message-dependent

The contents of the last five words of the message vary, depending on the message type. Although the standard GEM message is only eight words long, it's possible to send a longer, user-defined message to another task using the *appl_write( )* function. That's why the third word of the standard message format contains the number of additional bytes in the message. To receive the rest of the message, the task must read the message pipe directly, using the *appl_read( )* function, which will be discussed later on, along with appl_write( ).

There are 13 predefined AES messages, which the Screen Manager task sends to the application. These are as follows:

Message Number	Macro Name	Message Sent
10	MN_SELECTED	A menu item was selected by the user
20	WM_REDRAW	A window display needs to be redrawn
21	WM_TOPPED	The user has selected a new window to be active
22	WM_CLOSED	The user has clicked on the close box
23	WM_FULLED	The user has clicked on the full box
24	WM_ARROWED	The user has clicked on the scroll bars or arrows
25	WM_HSLID	The user wants to move the horizontal slider
26	WM_VSLID	The user wants to move the vertical slider
27	WM_SIZED	The user has dragged the size box
28	WM_MOVED	The user has dragged the move bar
29	WM_NEWTOP	A window has become active
40	AC_OPEN	A desk accessory has been selected from the menu
41	AC_CLOSE	An application has closed, and desk accessories have lost their handles

The macro definitions for the names of the various message ID's can be found in the header file GEMDEFS.H that comes with most C compilers. As you can see, 10 of the 13 start with the letters *WM*, which mean that they are window messages. Since these messages tell the other half of the window-management story started in the previous chapter, they will be discussed in detail here. The other three predefined messages, dealing with menus and desk accessories, will be covered along with those subjects later.

Window Display Refresh Messages

Perhaps the most important message your program can expect to get is WM_REDRAW. Most computer applications expect to update the screen display from time to time, to reflect changes in the information which the program outputs. But any application which takes advantage of the full range of GEM features must be ready to redraw the contents of each of its windows at any time.

Because of the nature of the GEM windowing system, all or part of a window might be covered or uncovered at any time. When that happens, the AES takes care of redrawing the window border areas, but it's up to the application to redraw the interior display of each of its affected windows.

You may expect to receive a WM_REDRAW message even if your application has only one window and that window has no window controls for moving or sizing it. The reason for this is that as long as your window includes a menu bar, you might start up a desk accessory that opens windows of its own. Your program can expect a window redraw message in any of the following circumstances:

- A new window is opened on the screen.
- Windows are reordered by sending a new window to the top of the stack as the active window.
- A window is made larger in any dimension.
- A window is moved from a position part way off the screen to a position where more of the window is on the screen.
- A window is closed, sized down, or moved, exposing a previously covered portion of another window. This window doesn't have to belong to your program. It may have been opened by another task, such as a desk accessory.
- A dialog is completed, and the dialog box is removed.

When these or similar events occur, the Screen Manager determines which portion of the screen display has been damaged, and notes the size and position of the rectangle enclosing this area. Then it checks each open window to see if any portion of the window's work area overlaps that rectangle. It then sends the application a separate redraw message for each window that needs to be refreshed. The format for this message is as follows:

Word Number	Contents
0	20 (WM_REDRAW), the message ID number
3	The handle of the window whose display needs refreshing
4	The *x* position of the damaged rectangle
5	The *y* position of the damaged rectangle
6	The width of the damaged rectangle
7	The height of the damaged rectangle

When your program receives the WM_REDRAW message, there is a set pattern of steps that you must take to restore the contents of that window. First, you must stabilize the state of the screen, so that no changes take place during the update process. Turn off the mouse with the *graf_mouse()* call:

```
graf_mouse(M_OFF, 0x0L);
```

The reason for hiding the mouse pointer is that the AES stores the image of the rectangle underneath the mouse and restores that image when the mouse is moved. If you merely overwrite the mouse with your graphics output, the next time the mouse is moved the system will restore the previous image, wiping out a rectangle of your new display. The second part of stabilizing the screen display is to lock the screen with the *wind_update()* call. The format for this call is

```
int status, code;
status = wind_update(code);
```

where *status* is equal to 0 if an error occurred. If there was no error, a nonzero quantity will be in status. The code value indicates the function of the call:

Code	Macro Name	Function
0	END_UPDATE	Notifies AES that the application is ending its window display update
1	BEG_UPDATE	Notifies AES that the application is beginning a window display update
2	END_MCTRL	Notifies AES that it should once more take control of the mouse when it leaves the active window area
3	BEG_MCTRL	Notifies AES that the application is taking control of all mouse functions, even when it moves out of the active window

The macro names for this function are defined in the file GEMDEFS.H. For purposes of starting the refresh process, use the call:

wind_update(BEG_UPDATE);

This call prevents the system from making display changes in the part of the screen being updated. Things could get very messy if a menu dropped down on top of your window while you were drawing in it.

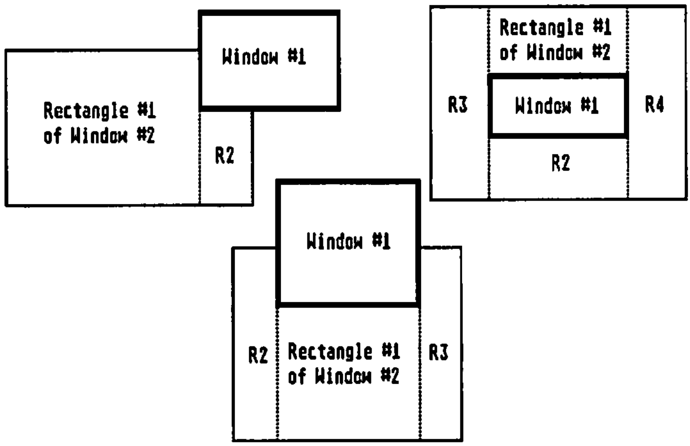
Next, comes the process known as "walking the rectangle list." Unless your window is the active (topmost) window, it's possible that part of it is covered by another window. The AES doesn't automatically limit your graphics output to the part of the window that's showing. Therefore, if your program redraws the entire window, it's going to destroy parts of other windows.

Keeping your graphics output within the visible portion of your window is strictly your program's responsibility. The AES helps you in this task, however, by keeping what's known as a *rectangle list.* When a window is partially obscured, GEM divides the visible portion of the window into the least possible number of nonoverlapping rectangles. For example, if two windows on the screen overlap at a corner, the visible portion of the top window will consist of one rectangle, while the visible portion of the bottom window will be divided into two rectangles. If the top window overlaps a side of the bottom window, the visible portion of the bottom window will be divided into three rectangles. If the top window is entirely contained within the bottom one, the visible part of the bottom window will be divided into four parts. As you increase the number of windows, the combinations increase as well. (See Figure 3-1.)

To find the list of visible rectangles for a particular window, you use the *wind_get( )* command. As you may remember from the previous chapter, this command contains two subcommands which are of interest here. One is WF_FIRSTXYWH, and the other is WF_NEXTXYWH.

WF_FIRSTXYWH returns the position and size of the first rectangle. WF_NEXTXYWH returns the position and size of the next rectangle in the list. Each subsequent call to wind_get with the WF_NEXTXYWH subcommand returns the position

Figure 3-1. Some Window Rectangle Possibilities for a Two-Window Screen



and size of the next rectangle of the list. When either type of wind_get( ) command returns a rectangle with a width and height of 0, you've reached the end of the list.

Now that you know what the rectangle list is, here is how to use it in your window refresh procedure. The next step is to get the first rectangle in the window's rectangle list with the wind_get( ) call:

wind_get
(wi_handle, WF_FIRSTXYWH, &wrec.g_x, &wrec.g_y,
&wrec.g_w, &wrec.g_h);

Now that you have the position and size of the damage rectangle, and the position and size of the first window rectangle, you must check to see if the two rectangles overlap anywhere. If they do, this third "overlap" rectangle marks the area whose display must be updated.

Figuring out the overlap area is fairly simple. The x position is equal to the greater of the two values of the original two rectangles, and the y position is equal to the greater of the two original y values. To find the width you first find the lesser of the two (x + width) values and then subtract this figure from the overlap x. To find the height, you take the lesser of the two (y + height) values and subtract it from the overlap *y*. In C code, the process of finding the overlap area looks like this:

```
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )

olapx = MAX(x1, x2);
olapy = MAX(y1, y2);
olapw = MIN(x1+w1, x2+w2) - olapx;
olaph = MIN(y1+h1, y2+h2) - olapy;
```

If both the overlap rectangle width and height are greater than 0, then there's an overlap area that needs redrawing. Before you do the actual redrawing, however, you must set a clipping rectangle. Clipping confines your graphics output to the specified rectangle. When the drawing operation attempts to go outside that rectangle, nothing is output. If your program uses the VDI to draw the window contents, you may set a clipping rectangle with the VDI call *vs_clip( )*. This call takes the form:

```
int handle, points[4];
vs_clip(handle, 1, points);
```

where *handle* is the VDI workstation handle, and *points* is an array containing the coordinates for two opposite corners of the rectangle. Note that the VDI representation of a rectangle as an array of points differs from the AES convention of specifying one point of origin and then specifying the width and height. To convert from AES format to VDI format, place the rectangle *x* and *y* in the first two array elements, *x* + width − 1 in the next and *y* + height − 1 in the last:

```
GRECT olap;
int handle, points[4];
    point[0] = olap.g_x
    point[1] = olap.g_y
    point[2] = olap.g_x + olap.g_w -1
    point[3] = olap.g_y + olap.g_h -1
    vs_clip(handle, 1, points);
```

As you will see in a later chapter, your program may also use the Object Library routine *objc_draw( )* to draw part of the window contents. The input parameters for this call include the position and size of a clipping rectangle. When you use this call, therefore, you may just give the dimensions of the overlap rectangle as the clip area.

Each subsequent clipping rectangle is treated in pretty much the same manner as the first. You get the next clipping rectangle with the call:

wind_get
(wi_handle, WF_NEXTXYWH, &wrec.g_x, &wrec.g_y,
&wrec.g_w, &wrec.g_h);

If the width and height of the rectangle aren't 0 (which would signify that you've already received the last rectangle on the list), you find the overlap of the window rectangle with the damage rectangle. If an overlap rectangle exists, you set the clipping rectangle and perform the redraw. This process continues until there are no more window rectangles to refresh. At that point, you unlock the screen display with a call to *wind_update( ):*

wind_update(END_UPDATE);

This lets the AES know that it's safe to drop menus and so on. If you forget this call, your program won't be able to access any menus. After unlocking the screen, you should turn on the mouse with a call to *graf_mouse( ):*

graf_mouse(M_ON, 0x0L);

To summarize, the steps to take when refreshing a window display are these:

1. Turn off the mouse with the graf_mouse( ) call.
2. Lock the screen display with the wind_update( ) call.
3. Get the first rectangle in the window's rectangle list with the wind_get( ) call.
4. If the width and height of this rectangle are greater than 0, calculate the size and position of the "overlap rectangle." This rectangle is made up of the area where the window rectangle overlaps the damage rectangle. If there are no more rectangles in the list (the width and height values are 0), go to step 7.
5. If the two rectangles did intersect, redraw the overlap rectangle. If the program uses the VDI to draw the window contents, convert the AES rectangle to an array of points and set a clipping rectangle with the VDI call vs_clip( ). If it uses an AES object tree to draw the window contents, set the clip area with the objc_draw( ) input parameters. If there was no overlap, go on without redrawing.

6. Get the next clipping rectangle with wind_get( ) and go back to Step 4.
7. Unlock the screen display with wind_update( ) and then turn the mouse back on with graf_mouse( ).

Handling the WM_REDRAW messages takes care of the case where system events force you to refresh your window. There are circumstances, however, under which you'll have to redraw the window even when you don't get a system message. For example, you don't get a redraw message for a window when you decrease its size, but you may want to change that window's contents all the same. Likewise, you don't get a redraw message when you move the slider in a scroll bar, but you'll want to change the window contents then, too. One way to perform the update is to simply call your window refresh routine. But another alternative is to send yourself a redraw message. When the program receives this message, the redraw is taken care of by the normal message-handler routine. The main advantage to sending yourself a message when you want to refresh a window is that the AES checks to see if there's already a redraw message waiting in the pipe. If there is, it "merges" the two requests by changing the damage rectangle to one large enough to include the two smaller damage rectangles. This helps prevent multiple refreshes. Such sequential redrawings slow down the program and give it an unprofessional, flickering appearance.

Sending and Receiving Messages

You send your program a message the same way you'd send a message to any other task. First, you create an eight-word message in a buffer. The standard format is used for this message. The first word contains the message ID, the second word contains the application ID of the task sending the message, the third contains the number of additional bytes (past the standard 16) used by the message, and the rest contain message-specific data. After creating the message, you send it by using the *appl_write( )* function. The syntax for this function is

```
int status, id, length, msgbuf[ ];
status = appl_write(id, length, msgbuf);
```

where *id* is the application ID of the task to which you are sending the message, *length* represents the length of the message in bytes (16 is the standard length for an eight-word message), and *msgbuf* is a pointer to the buffer which contains the message. The value returned in *status* equals 0 if there was an error in performing the function, and it's greater than 0 if no error occurred. To send a redraw message to your own application, therefore, you could use the following code:

```
int wi_handle                  /* The handle of the window to be
                                  redrawn */
GRECT r;                       /* The redraw rectangle */
int msg[8];                    /* The message buffer */
  msg[0] = WM_REDRAW;          /* Message type is window redraw
     message */
  msg[1] = gl_apid;            /* Application ID stored in Global
     array */
  msg[2] = 0;                  /* Message is standard 16 bytes
     long */
  msg[3] = wi_handle;          /* Handle of window to refresh */
  msg[4] = r.g_x;              /* Position and size of redraw
     rectangle */
  msg[5] = r.g_y;
  msg[6] = r.g_w;
  msg[7] = r.g_h;
  appl_write(gl_apid, 16, &msg);
```

Sending a message to your own application is easy, since a program can always find its own application ID by looking in the global array. To send a message to another application, however, you must first find its application ID, using the function *appl_find( )*. The syntax for this function is

```
int id;
char name[8];
id = appl_find(name);
```

where *name* is a null-terminated string containing the filename of the application. This string must be exactly eight characters long; if the filename is shorter, the end of the string should be padded with spaces to bring it to eight characters. The application's ID is returned in the variable *id*. If GEM can't find the application, a value of −1 is returned.

If you want to wait for a message event, you can use the *evnt_mesag( )* command to place the first message in your buffer. To read the message pipe directly, use the *appl_read( )* command. The format for this command is

```
int status, id, length, msgbuf[ ];
status = appl_read(id, length, msgbuf);
```

where *id* is the application ID for the application whose message pipe you're reading (generally your own). *Length* and *msgbuf* are the length of the message (in bytes) and a pointer to the message buffer, respectively. The function returns a status value of 0 if there was an error, and it's greater than 0 if there was no error. Generally you'll use the appl_read( ) function when you get a message whose third element contains a number greater than 0, signifying that there are more than 16 bytes in the message. You'll use the number of bytes specified in the third element for the length field in appl_read.

Messages for Moving, Sizing, or Closing a Window

In the previous chapter, the various window control boxes were mentioned, such as the size box and the full box. It was also mentioned that these boxes do not perform the indicated functions autonomously. In other words, if the user clicks the full box, the AES doesn't automatically resize the window all by itself. Instead, the Screen Manager sends a message to the application telling it what the user has done. When the program gets this message, it can either ignore it or honor it by changing the window using the *wind_set( )* function.

If you've created a window that includes the sizer control, your program should be prepared to deal with message 27, WM_SIZED. When the user moves the mouse to the size box, holds down the left mouse button, and drags the mouse, the Screen Manager displays an elastic image of a box that follows the mouse, indicating the new outlines of the window. When the user releases the mouse button, the Screen Manager erases the box, and sends the application message 27. The contents of the msgbuf message buffer after such a message is received looks like this:

Word Number	Contents
0	27 (WM_SIZED), the message ID number
3	The handle of the window that was requested to be sized
4	The requested *x* position of the window's left edge (the same as the current window *x* position)
5	The requested *y* position of the window's top edge (the same as the current window *y* position)
6	The requested width of the window
7	The requested window height

If you're willing to let the user size the window arbitrarily, you can just forward the window dimensions received in the message to the wind_set( ) command, which will resize the window to those dimensions:

wind_set
(msgbuf[3], WF_CURRXYWH, msgbuf[4], msgbuf[5], msgbuf[6], msgbuf[7]);

GEM itself constrains the sizing of a window to a limited degree. It won't allow a window larger than the screen, or so small that the scroll bar controls located in the window borders are totally obscured.

You may wish to set your own minimum and maximum size limits, or you may want your program to adjust the sizing request before passing it on to wind_set( ). For example, if you're working with a text-based application, you may want to limit window sizing to an even multiple of the default character-cell size or to even 16-bit boundaries. Printing graphics text is much faster when each line of text starts on a 16-bit boundary. Just remember that the rectangle returned by the WM_SIZED message describes the exterior dimensions of the window, including the border area. If your program is interested in controlling the interior or work area of the window, use the *wind_calc( )* function to convert the requested size from exterior to interior size, adjust that size, and then use wind_calc( ) to convert back before passing the dimensions to the wind_set( ) call.

A Redraw request is generated for a window when the wind_set( ) command is used to increase its size. In some cases, your program may want to reshuffle the display when the window is sized down as well. You can accomplish this either by calling the redraw routine directly or by sending yourself a redraw message after changing the size with wind_set( ).

Another way the user can indicate a desire to change the size of the window is by clicking on the fuller box. If the window controls flag for your window includes the fuller attribute, indicating that the window has a fuller box in the top right corner, your program should be prepared to handle message 23, WM_FULLED. The relevant items in the message buffer for this message are

Word Number	Contents
0	23 (WM_FULLED), the message ID number
3	The handle of the window whose full box was clicked

The fuller box is supposed to act like a toggle. That means if the window isn't at full size when the user clicks on it, it's supposed to expand to full size. If the window is already at full size when the user clicks, the window should contract to its previous size. As usual, it's up to you to implement this toggle mechanism. The AES helps out by keeping track of the window's current size, its maximum size, and its previous size. You may determine all three of these window dimensions by calling wind_get( ) with the appropriate subcommand (WF_CURRXYWH, WF_PREVXYWH, or WF_FULLXYWH). After you've found out the size and position of these rectangles, you can check to see if the current size equals the maximum size. If it doesn't, you may set the window to maximum size using wind_set( ). If it's already at that size, you may use wind_set( ) to change to the previous size. The subcommand to use is WF_CURRXYWH:

wind_set(wi_handle, WF_CURRXYWH, newx, newy, neww, newh);

Since you use wind_set( ) to change the window size in response to the WM_FULLED message, just as you do for the WM_SIZED message, the same rules about redraw messages apply. If you increase the window size, your program will get a redraw message. If you decrease its size, it won't get a redraw message. Therefore, if you want to redraw after making the window smaller, either call the redraw routine directly or send yourself a redraw message after changing the size with wind_set( ).

Not only may GEM windows be sized, but they may be moved as well. When the user places the mouse pointer on the drag bar, holds down the left button, and moves the mouse, the Screen Manager draws a dotted window outline that moves with the mouse. This outline shows the new window position. When the user lets go of the mouse button, the Screen Manager sends the program message 28, WM_MOVED. Any program that has a window with the mover attribute in its control word should be ready to handle the WM_MOVED message. The significant elements of this message are

Word Number	Contents
0	28 (WM_MOVED), the message ID number
3	The handle of the window whose move bar was dragged
4	The requested *x* position of the window's left edge
5	The requested *y* position of the window's top edge
6	The requested width of the window (the same as the current width)
7	The requested window height (the same as the current window height)

If you're willing to let the user move the window anywhere on the screen, you can merely forward the new window position to the wind_set command, using the WF_CURRXYWH subcommand:

wind_set
(msgbuf[3], WF_CURRXYWH, msgbuf[4], msgbuf[5], msgbuf[6], msgbuf[7]);

You will probably want to constrain the user's freedom to move windows around on the screen. The current version of GEM on the ST won't let you drag a window past the left or top screen borders, unless its starting position was beyond those borders. But it will let you move a window partially offscreen towards the bottom or to the right. Therefore, you may wish to prevent the user from moving the window past the right or bottom borders.

To keep the complete window display on screen, make its maximum *x* position equal to the width of the Desktop window minus the width of your application window, and its maximum y position equal to the height of the Desktop window minus the height of your application window.

Another constraint on window moving that would be valuable to your program would be to align the left edge of the window on a 16-bit word boundary. By rounding the coordinates for the left edge of the window to an even multiple of 16, you make it easier for GEM to move the window contents quickly, since it eliminates a considerable amount of bit-shifting and masking operations. Aligning an image to an even 8-byte or 16-byte boundary is known as *snapping*.

Usually when you use the CURRXYWH subcommand of wind_set( ) to move the window, the AES will perform a raster-copy operation that will move the window's contents automatically. The only time you'll get a refresh message for the moved window is if the window is partially offscreen, so that the AES doesn't have access to the complete contents of the window. If you prevent the user from moving the window partially offscreen, a move operation will never generate a redraw message for the window that you've moved.

When the AES wants your application to move one of its windows to the top of the screen and become the active window, it sends message 21, WM_TOPPED. This happens when the user selects a window to be active by clicking within its area, or when the current active window is closed by its application or desk accessory. The format for the WM_TOPPED message is

Word Number	Contents
0	21 (WM_TOPPED), the message ID number
3	The handle of the window the user clicked in

When you get this message, you should move the window to the top. The way to move a window to the top is with the WF_TOP subcommand of wind_set( ):

```
int wi_handle;
wind_set(wi_handle, WF_TOP, 0, 0, 0, 0);
```

Even if your application only has one window, it must always be ready to handle the WM_TOPPED message if it has a menu bar, since a desk accessory may open a second window. If no window is made active, you may not get any messages from the AES, and the user may be locked out of the program.

When the user clicks on the close box in the upper left corner of the window, the Screen Manager sends your application message 22, WM_CLOSED. The format for this message is

Word Number	Contents
0	22 (WM_CLOSED), the message ID number
3	The handle of the window whose close box was clicked

When you get this message, you should take whatever action is appropriate. Most of the time, you'll just close the window with *wind_close( ):*

wind_close(msgbuf[3]);

Sometimes, you'll want to put up an alert message like "Are you sure? [Yes] [No]" or "Save file before closing? [Yes] [No] [Cancel]" to make sure that the user doesn't accidently exit the program and lose valuable work. Other times, the WM_CLOSED message indicates that the user wants to move back a level. For example, when you close a folder display window in the Desktop application, the window doesn't disappear, but rather displays the contents of the next highest subdirectory. This kind of ambiguity can confuse the user, so use caution when defining the close box to mean anything other than getting rid of the window.

If you plan to allow the user to reopen the window, then you don't have to delete it immediately. But you should remember to delete it before you close the application.

Program 3-1, a C program, shows how to handle window messages that request you to redraw, size, full, close, or top a window.

Program 3-1. message.c

```
/************************************************************************/
/*                                                                      */
/*      MESSAGE.C -- Demonstrates the various window messages          */
/*      your program might receive, and how to handle them.            */
/*                                                                      */
/*                                                                      */
/************************************************************************/

#define FALSE 0
#define TRUE 1
#define APP_INFO ""
#define APP_NAME "First Window"
#define WDW_CTRLS (NAME|CLOSER|SIZER|MOVER|FULLER)
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )
```

```
#include "aesshell.c"

int wh2, msg[8];  /* second window handle, message buffer */

demo()
{
  int closed=0;

   wh2=wind_create(WDW_CTRLS, desk.g_x, desk.g_y, desk.g_w, desk.g_h);
                              /* Create another window */
   wind_set(wh2,WF_NAME, "Second Window",0,0);
                              /* set name for window */
   wind_open(wh2, desk.g_x + desk.g_w/4, desk.g_y + desk.g_h/4,
            desk.g_w/2, desk.g_h/2);
                              /* open the window to half size */


    do                                  /* main program loop */
    {
        evnt_mesage(&msg);              /* get messages... */
        closed += handle_msg();         /* and handle them... */
     }
    while(closed<2);                    /* til both windows are closed */

    wind_open(wi_handle,0,0,0,0);       /* open 1st wdw so shell can close */
    wind_delete(wh2);                   /* delete second window */

}

handle_msg()                          /* message handler */
{
   int closed = FALSE;

   switch(msg[0])                    /* check message type */
      {
      case WM_REDRAW:                /* if redraw, call refresh routine */
         refresh(msg[3], (GRECT *)&msg[4]);
         break;

      case WM_TOPPED:                /* if topped, send to top */
         wind_set(msg[3], WF_TOP, 0, 0, 0, 0);
         break;

      case WM_SIZED:                 /* if sized, check for min size,
                                          then resize */
         msg[6] = MAX(msg[6], cellw*8);
         msg[7] = MAX(msg[7], cellh*4);
         wind_set(msg[3], WF_CURRXYWH, msg[4], msg[5], msg[6], msg[7]);
         redraw_msg(msg[3], (GRECT *)&msg[4]);
         break;

       case WM_MOVED:               /* if moved, make sure the window
                                          stays on the Desktop */
         if (msg[4] + msg[6] > desk.g_x + desk.g_w)
             msg[4] = desk.g_x + desk.g_w - msg[6];

         if (msg[5] + msg[7] > desk.g_y + desk.g_h)
             msg[5] = desk.g_y + desk.g_h - msg[7];

         wind_set(msg[3], WF_CURRXYWH, msg[4], msg[5], msg[6], msg[7]);
         break;

       case WM_FULLED:              /* if fulled, do toggle routine */
         toggle(msg[3]);
         break;
```

```
        case WM_CLOSED:         /* if closed, close window and
                                   increment count */
           wind_close(msg[3]);
           closed = TRUE;
           break;

        default:
          break;
        }
     return(closed);
}

toggle(wh)      /*  routine to handle WM_FULLED message */
   int wh;
{
   GRECT prev, curr, full;

   /* get current, previous, and full size for window */
   wind_get(wh, WF_CURRXYWH, &curr.g_x, &curr.g_y, &curr.g_w, &curr.g_h);
   wind_get(wh, WF_PREVXYWH, &prev.g_x, &prev.g_y, &prev.g_w, &prev.g_h);
   wind_get(wh, WF_FULLXYWH, &full.g_x, &full.g_y, &full.g_w, &full.g_h);

   /* If full, change to previous (unless that was full also) */
   if(((curr.g_x == full.g_x) &&
       (curr.g_y == full.g_y) &&
       (curr.g_w == full.g_w) &&
       (curr.g_h == full.g_h))     &&
     ((prev.g_x != full.g_x) ||
       (prev.g_y != full.g_y) ||
       (prev.g_w != full.g_w) ||
       (prev.g_h != full.g_h)))
      {
      wind_set(wh, WF_CURRXYWH, prev.g_x, prev.g_y, prev.g_w, prev.g_h);
      redraw_msg(wh, &prev); /* send a redraw message, cause AES won't */
      }

   /* If not full, change to full */
   else
      wind_set(wh, WF_CURRXYWH, full.g_x, full.g_y, full.g_w, full.g_h);

}


refresh(wh, drect)  /* routine to handle window_refresh (WM_REDRAW) */
   int    wh;          /*  window handle from msg[3] */
   GRECT *drect;       /*  pointer to damage rectangle  */
   {
   GRECT   wrect;   /*  the current window rectangle in rect list */

   graf_mouse(M_OFF, 0L);       /* turn off mouse */
   wind_update(BEG_UPDATE);     /* lock screen */

   wind_get                     /* get first rectangle */
    (wh, WF_FIRSTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w, &wrect.g_h);

   while ( wrect.g_w && wrect.g_h ) /* while not at last rectangle,  */
      {
      if (overlap(drect, &wrect))   /* check to see if this one's damaged, */
         {
             set_clip(&wrect);      /* if it is, set clip rectangle */
             display();             /* redraw, and turn clip off */
             vs_clip(handle, FALSE, (int *)&wrect );
         }
      wind_get(wh, WF_NEXTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w,
         &wrect.g_h);
      }
```

```
   wind_update(END_UPDATE);          /* unlock screen */
   graf_mouse(M_ON, 0x0L);           /* turn mouse pointer back on */
   }


display()          /* draw the window display */
{
  int points[4];  /* VDI points array */

  wind_get(msg[3], WF_WORKXYWH, &work.g_x, &work.g_y,
          &work.g_w, &work.g_h);     /* find work area */
  clear_rect(&work);                 /* and clear it */

  grect_conv(&work, &points);        /* convert work grect to array */
  vsf_interior(handle,2);            /* set fill type to pattern */
  vsf_style(handle, 7 * msg[3] + 2); /* adjust fill pattern */
  vsf_color(handle, msg[3]);         /* set color */
  v_ellipse(handle, points[0] + (work.g_w/2), points[1] + (work.g_h/2),
            work.g_w/2, work.g_h/2); /* draw a filled ellipse */

}


/* >>>>>>>> Utility routines used by other functions <<<<<<<<<<<<< */


set_clip(r)    /* set clip to specified rectangle  */
   GRECT   *r;
   {
   int  points[4];

   grect_conv(r, points);
   vs_clip(handle, TRUE, points);
   }


overlap(r1, r2)       /* compute overlap of two rectangles   */
   GRECT   *r1, *r2;
   {
   int x, y;

   x = MAX(r2->g_x, r1->g_x);
   y = MAX(r2->g_y, r1->g_y);
   r2->g_w = MIN(r2->g_x + r2->g_w, r1->g_x + r1->g_w) -x;
   r2->g_h = MIN(r2->g_y + r2->g_h, r1->g_y + r1->g_h) -y;
   r2->g_x = x;
   r2->g_y = y;
   return( (r2->g_w > 0) && (r2->g_h > 0) );
   }


redraw_msg(wh, r) /* Send Redraw Message to your own window  */
   int   wh;
   GRECT   *r;
   {
   int  msg[8];

   msg[0] = WM_REDRAW;
   msg[1] = gl_apid;
   msg[2] = 0;
   msg[3] = wh;
   msg[4] = r->g_x;
   msg[5] = r->g_y;
   msg[6] = r->g_w;
   msg[7] = r->g_h;
   appl_write(gl_apid, 16, &msg);
   }
```

Program 3-2 is an abbreviated version of the same program in machine language.

Program 3-2. message.s

```
*************************************************************
*                                                           *
*     MESSAGE.S Shows how to handle  window messages        *
*                                                           *
*************************************************************

*** External references

** Export:

     .xdef    demo     * external demo subroutine.
     .xdef    wdwctrl
     .xdef    wdwtitl
     .xdef    wdwinfo

** Import:

     .xref    vdi
     .xref    aes

     .xref    contrl0  * all of the VDI data arrays
     .xref    contrl1
     .xref    contrl2
     .xref    contrl3
     .xref    contrl4
     .xref    contrl5
     .xref    contrl6
     .xref    intin
     .xref    ptsin
     .xref    ctrl0  * all of the AES data arrays
     .xref    ctrl1
     .xref    ctrl2
     .xref    ctrl3
     .xref    ctrl4
     .xref    aintin
     .xref    aintout
     .xref    addrin

     .xref    deskx
     .xref    desky
     .xref    deskw
     .xref    deskh
     .xref    workx
     .xref    worky
     .xref    workw
     .xref    workh

       .text

     demo:
        move     #0,d4  * close window flag in d4

* create and open second window

        move     #100,ctrl0  * command = wind_create
        move     #5,ctrl1    * 5 input integers
        move     #1,ctrl2    * 1 ouput integer
        move     wdwctrl,aintin * window ctrl flag
        move     deskx,aintin+2 * max x
        move     desky,aintin+4 * max y
        move     deskw,aintin+6 * max width
        move     deskh,aintin+8 * max height
```

```
        jsr       aes
        move      aintout,wh2     * save window handle

*** set window name

        move      #105,ctrl0  * command = wind_set
        move      #5,ctrl1    * 5 input integers
        move      wh2,aintin  * window handle
        move      #2,aintin+2     * subcommand = set window name
        move.l    #titl2,aintin+4 * point to title

        jsr       aes


*** Open the window

        move      #101,ctrl0  * command = wind_open
        move      #5,ctrl1
        move      #1,ctrl2    * 1 output integers
        move      deskw,d0
        asr       #2,d0
        add       deskx,d0
        move      d0,aintin+2   * initial x = deskw/4 + deskx
        move      deskh,d0
        asr       #2,d0
        add       desky,d0
        move      d0,aintin+4   * initial y = deskh/4 + desky
        move      deskw,d0
        asr       #1,d0
        move      d0,aintin+6   * initial width = deskw/2
        move      deskh,d0
        asr       #1,d0
        move      d0,aintin+8   * initial height = deskh/2

        jsr       aes


     main:
        move      #23,ctrl0     * opcode = evnt_messag
        move      #0,ctrl1
        move      #1,ctrl2      * 1 intout
        move      #1,ctrl3      * 1 addrin
        move      #0,ctrl4
        move.l    #msg,addrin
        jsr       aes
        move      #0,ctrl3

        jsr       msghand       * handle the message
        cmpi      #0,d4         * check if window close
        beq       main          * if not, keep going

* delete window 2

        move      #103,ctrl0  * command = wind_delete
        move      #1,ctrl1
        move      #1,ctrl2
        move      wh2,aintin

        jmp       aes
** >>>>>>>>>> End of Main Program Code <<<<<<<<<<<<<<< **

*** Message handler subroutine ****
     msghand:
        move      msg,d5        * check message type
        cmpi      #27,d5        * WM_SIZED?
        bgt       msg5          * if greater, skip
        bne       msg2
```

```
    msg1:
       move      #105,ctrl0       * command = wind_set
       move      #6,ctrl1         * 6 input integers
       move      msg+6,aintin     * window handle
       move      #5,aintin+2      * subcommand = set current size
       move      msg+8,aintin+4
       move      msg+10,aintin+6
       move      msg+12,aintin+8
       move      msg+14,aintin+10
       jmp       aes

    msg2:
       cmpi      #22,d5        * WM_CLOSED?
       bne       msg3

* Close the Window

       move      #102,ctrl0  * command = wind_close
       move      #1,ctrl1
       move      #1,ctrl2
       move      #0,ctrl3
       move      #0,ctrl4
       move      wh2,aintin

       move      #1,d4
       jmp       aes

    msg3:
       cmpi      #21,d5        * WM_TOPPED?
       bne       msg4

       move      #105,ctrl0  * command = wind_set
       move      #6,ctrl1    * 6 input integers
       move      msg+6,aintin    * window handle
       move      #10,aintin+2    * subcommand = WF_TOP

       jmp       aes
    msg4:
       cmpi      #20,d5        * WM_REDRAW?
       bne       msg5
       jsr       refresh

    msg5:
       rts

*** Window refresh subroutine ***
    refresh:
* turn mouse off

       move      #78,ctrl0    * command = graf_mouse
       move      #1,ctrl1     * 1 input integers
       move      #1,ctrl2     * 1 output integer
       move      #256,aintin  * hide the mouse

       jsr       aes

* lock scren

       move      #107,ctrl0   * command = wind_update
       move      #1,ctrl1     * 1 input integers
       move      #1,ctrl2     * 1 output integer
       move      #1,aintin    * code = BEG_UPDATE

       jsr       aes
```

```
* Find first window rectangle

        move    #104,ctrl0   * command = wind_get
        move    #2,ctrl1     * 2 input integers
        move    #5,ctrl2     * 5 ouput integers
        move    msg+6,aintin * window handle
        move    #11,aintin+2 * WF_FIRSTxYWH command

        jsr     aes

    refresh1:                       * check for empty rectangle
        move    aintout+6,d0
        or      aintout+8,d0
        beq     refresh3            * if empty, at end, so quit

        move    msg+8,d0
        move    aintout+2,d1
        cmp     d0,d1               * x = MAX (x1, x2)
        bcs     olap1
        move    d1,d0               * overlap x is in d0

    olap1:
        move    msg+10,d1
        move    aintout+4,d2
        cmp     d1,d2
        bcs     olap2
        move    d2,d1               * overlap y is in d1

    olap2:
        move    msg+8,d2
        add     msg+12,d2
        move    aintout+2,d3
        add     aintout+6,d3
        cmp     d2,d3
        bhi     olap3
        move    d3,d2           * d2 = MIN(x1+w1, x2+w2)

    olap3:
        sub     d0,d2
        move    d2,aintout+6    * overlap w = d2 - overlap x

        move    msg+10,d2       * d2 = y1
        add     msg+14,d2       * + h1
        move    aintout+4,d3    * d3 = y2
        add     aintout+8,d3    * + h2
        cmp     d2,d3
        bhi     olap4
        move    d3,d2           * d2 = MIN(y1+h1, y2+h2)

    olap4:
        sub     d1,d2
        move    d2,aintout+8    * overlap h = d2 - overlap y

        or      aintout+6,d2    * are width and height both 0?
        beq     refresh2        * if so, skip redraw and get next rect

* set clip rectangle

        move    #129,contrl0    * opcode for set clip (vs_clip)
        move    #2,contrl1      * two points in ptsin
        move    #0,contrl2
        move    #1,contrl3
        move    #0,contrl4
        move    #1,intin        * turn clipping on
```

```
        move    d0,ptsin          * points[0] = overlap x
        add     aintout+6,d0
        subq    #1,d0             * points[2] = overlap x+w -1
        move    d0,ptsin+4
        move    d1,ptsin+2        * points[1] = overlap y
        add     aintout+8,d1
        subq    #1,d1
        move    d1,ptsin+6        * points[3] = overlap y+h -1

        jsr     vdi

* redraw the display
        jsr display

* turn clipping off

        move    #129,contrl0      * opcode for set clip (vs_clip)
        move    #2,contrl1        * two points in ptsin
        move    #0,contrl2
        move    #1,contrl3
        move    #0,contrl4
        move    #0,intin          * turn clipping off

        jsr     vdi

* get next window rectangle

    refresh2:
        move       #104,ctrl0   * command = wind_get
        move       #2,ctrl1     * 2 input integers
        move       #5,ctrl2     * 5 ouput integers
        move       msg+6,aintin * window handle
        move       #12,aintin+2 * WF_NExTxYWH command

        jsr        aes
        bra        refresh1

* unlock screen

    refresh3:
        move       #107,ctrl0   * command = wind_update
        move       #1,ctrl1     * 1 input integers
        move       #1,ctrl2     * 1 output integer
        move       #0,aintin    * code = END_UPDATE

        jsr        aes

* turn mouse on

        move       #78,ctrl0    * command = graf_mouse
        move       #1,ctrl1     * 1 input integers
        move       #1,ctrl2     * 1 output integer
        move       #257,aintin  * hide the mouse

        jmp        aes


*** Window display subroutine ***

    display:
* Find window work area size

        move       #104,ctrl0   * command = wind_get
        move       #2,ctrl1     * 2 input integers
        move       #5,ctrl2     * 5 ouput integers
        move       msg+6,aintin
        move       #4,aintin+2     * WF_WORKXYWH command
```

```
        jsr     aes
        move    aintout+2,workx   * store work x,y,w,h
        move    aintout+4,worky
        move    aintout+6,workw
        move    aintout+8,workh

* set fill pattern to hollow

        move    #23,contrl0     * opcode for set fill type
        move    #0,contrl1
        move    #0,contrl2
        move    #1,contrl3      * one integer in intin
        move    #1,contrl4
        move    #0,intin        * select hollow fill type

        jsr     vdi

*** Storage space and data constants
        .data
        .even
        msg:    .ds.w 8
        wdwtitl:    .dc.b 'First Window',0
        titl2:    .dc.b 'Second Window',0
        wdwinfo:    .dc.b '',0
        wdwctrl:   .dc.w 35
        wh2:    .ds.w
        .end
```

Scroll Bars and Their Messages

When you want to display more information than you can fit onscreen at one time, you may wish to make the window a view port onto a larger area. By using slide bars (or *scroll bars*), you allow the user to select which portion of the window's contents to view. As with the other GEM features discussed here, actually scrolling the document is the responsibility of your program, but GEM helps out by providing a standard framework within which scrolling may be implemented.

The standard GEM scroll bar consists of three elements: a movable bar called a slider, a long rectangular box the slider moves within, and arrow characters at either end of the rectangular box. These elements can be added to the window by specifying the elements LFARROW, HSLIDE, RTARROW and/or the elements UPARROW, VSLIDE, DNARROW as part of the window controls flag used by wind_create( ).

When a window contains all three elements (for instance, the LFARROW, HSLIDE, and the RTARROW), the user can take three types of action. First, the user can drag the slider by positioning the mouse pointer over it, holding the left button down, and moving the mouse. Second, the user can click on either arrow, indicating that the contents of the window are to be moved a character at a time. Or, third, the user can click on the scroll bar between the slider and the arrow, indicating that the contents of the window are to be moved a page at a time. All of these events generate window messages. These messages will be discussed in detail below.

In order to maintain slider bars in a GEM application, you must perform three tasks.

Slider size. First, you must keep track of the slider size. The portion of the scroll bar that's filled by the slider bar is supposed to represent the percentage of the total display area shown on screen. If half of the total display area is shown onscreen at a time, the slider bar should fill half the scroll bar. The AES allows you to set the size of the bar with a value in the range 1–1000. Each unit corresponds to one tenth of 1 percent of the total size of the scroll bar. A bar size of 1 fills the minimum possible area, while a bar size of 1000 fills the entire slide box. If the length of the part of the display that's seen is less than the total length available for display, you can calculate the size of the slider with the formula:

```
size = 1000 * (length_seen/total_length)
```

If you use this formula, you must be careful to check that the length seen is less than the total length, however, or you'll come up with a number larger than 1000. Also, you should probably use 32-bit variables for the computation to avoid exceeding the size limit of 16-bit integers. In C, these are variables that are declared to be of the type long. After you've computed the relative size of the slider, you can set the bar to this size using the WF_VSLSIZE or WF_HSLSIZE *wind_set()* function:

```
wind_set(wi_handle, WF_VSLSIZE, size, 0, 0, 0);
wind_set(wi_handle, WF_HSLSIZE, size, 0, 0, 0);
```

Your program should adjust the size of the slider if either the window or the document changes size. If the size of the window changes, you'll know because you will have received the WM_SIZED message and will have sized the window with wind_set( ).

When the size of the document changes, because the user deleted a line of text, for example, you should update the slider size only if the change alters the proportions significantly. After calculating the new size, check it against the current size. You can find the current slider size by using the

WF_HSLSIZE and WF_VSLSIZE subcommands of *wind_get( ):*

wind_get(wi_handle, WF_VSLSIZE, &size, &dummy, &dummy, &dummy);

wind_get(wi_handle, WF_HSLSIZE, &size, &dummy, &dummy, &dummy);

where *size* is the variable in which the current size is returned, and *dummy* is a dummy place-holder variable which must be used because wind_get( ) returns four values. If the new size is equal to the old size, you needn't update it.

Slider position. The second part of the task of maintaining scroll bars is keeping track of the slider position.

As with slider size, the slider position setting has a range of 1–1000. But this setting is tricky. It doesn't reflect the absolute position of the slider within the scroll bar. Instead, it marks the position of the top of the slider bar relative to its possible range of positions. Since the slider itself can take up a significant portion of the scroll bar, the top of the bar usually can't go down to the bottom of the scroll bar. For example, if the window display shows 20 lines of a 200-line text document, the vertical slider fills 10 percent of the area of the scroll bar. That means the top of the slider can never go down more than 90 percent of the way. The other 10 percent is taken up by the bar. In that situation, position 1000, the bottommost position on the bar, is 90 percent of the way down the scroll bar.

If the window displays 20 lines of a 100-line document, the slider occupies 20 percent of the area of the scroll bar. In that case, position 1000, the farthest that the top of the slider can go, is only 80 percent of the way down the scroll bar. Therefore, you must calculate the position of the top of the slider as a fraction of the total available range of motion. For example, in the case of the 200-line document, 20 of which are displayed, the total range is 200 − 20, or 180 lines. If line 39 is displayed in the top of the window, the bar should be placed at position (1000 * 39 / 180), or 216. This position setting is a little larger than the setting of 200 you'd intuitively guess to be the correct setting when the top of the display shows the line that's 20 percent of the way down the document. The larger the slide bar gets, the larger this discrepancy grows. For example, if you make the document half as long, you'd expect the setting to double. But if the window shows 20 lines of a 100-line document, and the top of the window shows line 39, the correct position setting is (1000 * 39 / 80), or 487, which is significantly more than double the earlier setting of 216. Where the portion of the display shown in the window is less than the total amount available for display, you can calculate the slider position with this formula:

position = 1000 * window_start / (total_length − length_seen)

where *window_start* represents the position that's displayed at the beginning of the window. For example, if line 40 of a document were displayed at the top of the window, window_start would be 39, because it's conventional to start counting the display position at 0. That way, if you're at the top of the document, the slider position is 0 (0 divided by any number is always 0), rather than some small fractional position number. Remember also that if the length seen is equal to or greater than the total length (the slider size is 1000), you must always position the slider at 0. Once you've calculated the correct slider position, you position it with the WF_HSLIDE or WF_VSLIDE subcommand of wind_set( ):

```
wind_set(wi_handle, WF_VSLIDE, position, 0, 0, 0,);
wind_set(wi_handle, WF_HSLIDE, position, 0, 0, 0,);
```

You'll know it's time to update the position of the slider when you get one of the window messages associated with the scroll bar. For example, when the horizontal slider is moved, message 25, WM_HSLID is sent:

Word Number	Contents
0	25 (WM_HSLID), the message ID number
3	The handle of the window whose horizontal slider was dragged
4	The requested position for the left edge of the slider (a number from 0 to 1000, where 0 = far left, 1000 = far right)

The user, dragging the vertical slider, generates message 26, WM_VSLID:

Word Number	Contents
0	26 (WM_VSLID), the message ID number
3	The handle of the window whose vertical slider was dragged
4	The requested position for the top edge of the slider (a number from 0 to 1000, where 0 = top, 1000 = bottom)

Since the number returned in msgbuf[4] is in the proper format, you may want to pass the new setting on to wind_set( ) unchanged:

wind_set(msgbuf[3], WF_VSLIDE, msgbuf[4], 0, 0, 0);

If the display is comprised of indivisible logical units, however, you'll want to round the movement of the slider to the nearest such unit. If the window display shows lines of text, for example, you'll want to scroll the display by an even number of text characters. One easy way of rounding is to find the window start position that corresponds to slider position and convert that start position back to a slider position. How to calculate the window start position will be discussed under the section on refreshing the window display later in this chapter.

When any of the other events associated with slide bars occurs, the Screen Manager sends message 24, WM_ARROWED:

Word Number	Contents
0	24 (WM_ARROWED), the message ID number
3	The handle of the window whose scroll bar was clicked
4	The action requested by the user: 0 Page up (user clicked on scroll bar above vertical slider) 1 Page down (user clicked on scroll bar below vertical slider) 2 Line up (user clicked on up arrow) 3 Line down (user clicked on down arrow) 4 Page left (user clicked on scroll bar left of horizontal slider) 5 Page right (user clicked on scroll bar right of horizontal slider) 6 Column left (user clicked on left arrow) 7 Column right (user clicked on right arrow)

As you can see, the eight possible cases are represented as subcommands in msgbuf[4]. What exactly your program does in response to these messages depends somewhat on the type of information being displayed. By convention, when the user clicks on an arrow, the display should scroll by the smallest indivisible unit. If the display is text, for example, you increment or decrement the starting position of the display by one character row or column and move the slider accordingly. When the user clicks on the scroll bar between an arrow and the slider, you should scroll the display by a larger increment, typically the number of units that fit on a display screen. Thus, if 20 lines of text fit on a screen, you might scroll the document by 20 lines. Check first to make sure that you're more than 20 lines from the beginning or end of the document.

In addition to window messages, your program should be ready to adjust the slider position in response to changes in the window size or document size. As with the slider size, however, before you make any change in the position of the slider, you should first check to make sure that the new position will actually be different than the current one. You can determine the current slider position with the WF_HSLIDE and WF_VSLIDE subcommands of wind_get( ):

```
wind_get(wi_handle, WF_VSLIDE, &position, &dummy,
   &dummy, &dummy);
wind_get(wi_handle, WF_HSLIDE, &position, &dummy,
   &dummy, &dummy);
```

where *position* is the variable in which the current position is returned, and *dummy* is a dummy place-holder variable which must be used because wind_get( ) always returns exactly four values. If the new position is equal to the old one, you needn't perform the update.

Updating the window display. The third and final part of maintaining a window that contains scroll bars is updating the window display. You will not get a window redraw message when you use wind_set to change the size or position of the slider. You must take the initiative either to call your redraw routine directly, or to send yourself a redraw message. In order to calculate where in the document to start the window display, you may have to do the reverse of the calculation that you did to find the slider position from the document start:

```
window_start=slider_position*(total_length-length_seen)/1000
```

Again, window_start counts the top of the document as 0, and you'll probably want to use 32-bit variables in order to avoid exceeding the limits of 16-bit arithmetic. Once you have the window_start position, you'll want to save it. That way, if the user clicks on one of the arrows, you can simply increment or decrement it by 1 and redraw, and if the window size changes, you can begin the window in the same place.

Program 3-3 shows how to handle the messages associated with maintaining the scroll bar.

Program 3-3. scroll.c

```
/*************************************************************************/
/*                                                                       */
/*      SCROLL.C -- Demonstrates how to manage scroll bars              */
/*      in a window.                                                    */
/*                                                                       */
/*                                                                       */
/*************************************************************************/

#define FALSE 0
#define TRUE 1
#define APP_INFO ""
#define APP_NAME "Scroll Window Example"
#define WDW_CTRLS (NAME|CLOSER|VSLIDE|UPARROW|DNARROW)
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )

#define TROWS 28    /* total number of text row */
#define TCOLS 30    /* total number of text columns */
#define SROWS 9     /* number of rows seen at a time */
#include "aesshell.c"

int  msg[8];   /* message buffer */
int ttop;      /* the top line of text currently in the window */

char *text[TROWS] =
     {
     "This is a sample help window. ",
     "The total text is too large to",
     "be displayed on screen at one ",
     "time. In order to see the rest",
     "of the text, you must use the ",
     "right scroll bar.             ",
     "                              ",
     "There are three different ways",
     "you can scroll the text. The  ",
     "first is to click on the up or",
     "down arrows.  This scrolls the",
     "text up or down, one line.    ",
     "Of course, if the first line  ",
     "of text is already at the top,",
     "you can't scroll up.  Nor can ",
     "you scroll down if you're at  ",
     "the botton of the window.     ",
     "                              ",
     "The second way to scroll is to",
     "click on the space between the",
     "slider and the arrow.  This   ",
     "moves the text up or down a   ",
     "page at a time.               ",
     "                              ",
```

```
    "The third way is to drag the  ",
    "slider with the mouse.  This  ",
    "moves the text a proportional ",
    "distance up or down.          ",
    };

demo()
{
  int done;            /* close window flag */
  long slsize;         /* size of slider */

    vst_alignment(handle, 0, 5, &done, &done); /* set text alignment */
    ttop = 0;  /* start with top line of text at top of window */

     /* calculate exterior size of window for text */

  wind_calc(0, WDW_CTRLS,
          cellw * 5, cellh * 5, cellw * TCOLS+4, cellh * SROWS+4,
          &work.g_x, &work.g_y, &work.g_w, &work.g_h);

     /* set the window to that size */

  wind_set(wi_handle, WF_CURRXYWH,
          work.g_x, work.g_y, work.g_w, work.g_h);

     /* find slider size and set it */

  slsize = (1000 * SROWS) / TROWS;
  wind_set(wi_handle, WF_VSLSIZE, (int)slsize, 0, 0, 0);

  wind_get(wi_handle, WF_WORKXYWH, &work.g_x, &work.g_y,
          &work.g_w, &work.g_h);    /* find work area */

    do                             /* main program loop */
    {
          evnt_mesag(msg);         /* get message ... */
          done = handle_msg();     /* and handle them ... */
          }
    while(!done);                  /* til window is closed */


}

handle_msg()                    /* message handler */
{
   int done = FALSE;
   long temp;

   switch(msg[0])              /* check message type */
      {
      case WM_REDRAW:          /* if redraw, call refresh routine */
         refresh(msg[3], (GRECT *)&msg[4]);
         break;

      case WM_TOPPED:          /* if topped, send to top */
         wind_set(msg[3], WF_TOP, 0, 0, 0, 0);
         break;

       case WM_CLOSED:        /* if closed, set flag  */
          done = TRUE;
          break;

       case WM_VSLID:         /* slide bar was dragged */
          temp = msg[4] * (TROWS-SROWS) / 1000; /* calc ttop */
          ttop = temp;
          refresh(wi_handle, &work);     /* redraw window */
          move_slide();                  /* and move slider */
          break;
```

```
        case WM_ARROWED:        /* arrow was clicked */
          switch (msg[4])
          {
          case 0:  /*page up*/
             ttop = MAX(0, ttop-SROWS);
             break;
          case 1:  /*page down*/
             ttop = MIN(TROWS-SROWS, ttop+SROWS);
             break;
          case 2:  /* row up */
             ttop = MAX(0, ttop-1);
             break;
          case 3:  /* row down */
             ttop = MIN(TROWS-SROWS, ttop+1);
             break;
          default:
             break;
          }
          refresh(wi_handle, &work);  /* redraw window */
          move_slide();               /* move the slider */
          break;

       default:
         break;
       }
    return(done);
}

refresh(wh, drect)  /* routine to handle window_refresh (WM_REDRAW) */
   int   wh;        /*  window handle from msg[3] */
   GRECT *drect;    /*  pointer to damage rectangle  */
   {
   GRECT   wrect;   /*  the current window rectangle in rect list */

   graf_mouse(M_OFF, 0L);       /* turn off mouse */
   wind_update(BEG_UPDATE);     /* lock screen */

   wind_get                     /* get first rectangle */
    (wh, WF_FIRSTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w, &wrect.g_h);

   while ( wrect.g_w && wrect.g_h ) /* while not at last rectangle,   */
      {
      if (overlap(drect, &wrect))   /* check to see if this one's damaged, */
        {
           set_clip(&wrect);        /* if it is, set clip rectangle */
           display();               /* redraw, and turn clip off */
           vs_clip(handle, FALSE, (int *)&wrect );
        }
      wind_get(wh, WF_NEXTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w,
        &wrect.g_h);
      }

   wind_update(END_UPDATE);         /* unlock screen */
   graf_mouse(M_ON, 0x0L);          /* turn mouse pointer back on */
   }


display()          /* draw the window display */
{
  int x;

/* print each visible line of text, starting with ttop */

  for(x = 0; x < SROWS; x++)
      v_gtext(handle, work.g_x, cellh * x + work.g_y, text[ttop+x]);
}
```

```
move_slide()  /* move the slider to match ttop */
{
   int cslide, nslide;
   long temp;

   wind_get(wi_handle, WF_VSLIDE, &cslide, &nslide, &nslide, &nslide);
   temp = 1000 * ttop/ (TROWS-SROWS);
   if ( (nslide=temp) != cslide)
      wind_set(wi_handle, WF_VSLIDE, nslide, 0, 0, 0);
}


/* >>>>>>>> Utility routines used by other functions <<<<<<<<<<<<<< */


set_clip(r)   /* set clip to specified rectangle  */
   GRECT  *r;
   {
   int  points[4];

   grect_conv(r, points);
   vs_clip(handle, TRUE, points);
   }


overlap(r1, r2)      /* compute overlap of two rectangles   */
   GRECT   *r1, *r2;
   {
   int x, y, x1, y1;

   x = MAX(r2->g_x, r1->g_x);
   y = MAX(r2->g_y, r1->g_y);
   r2->g_w = MIN(r2->g_x + r2->g_w, r1->g_x + r1->g_w) -x;
   r2->g_h = MIN(r2->g_y + r2->g_h, r1->g_y + r1->g_h) -y;
   r2->g_x = x;
   r2->g_y = y;
   return( (r2->g_w > 0) && (r2->g_h > 0) );
   }
```

Notice that the window display was updated by printing each line of text, one after the other, using the *v_gtext()* routine. This process is fairly quick, as long as you keep the text aligned on byte boundaries. When speed is important, however, you'll want to update a scrolled display by moving the block whose contents remain valid with the VDI raster routine *vro_cpyfm()* and then filling in the new data. In some cases, you may even wish to keep an entire copy of the window's contents stored offscreen in a memory buffer area. Then, when you want to refresh the window, you only have to move the relevant portion of that information using vro_cpyfm( ).

Mouse Button Events

The next type of event is the mouse button event. This function does much more than merely wait for the user to push a button. It allows you to specify the particular combination of buttons you are waiting for, whether you want those buttons to be pressed or released, and the number of button clicks you want to wait for as well. To wait for a mouse button event, call *evnt_button:*

```
int clicked, clicks, bmask, bstate, mousex, mousey, button, kstate;
    clicked = evnt_button(clicks, bmask, bstate, &mousex,
        &mousey, &button, &kstate);
```

The input parameter *clicks* lets the AES know the maximum number of button clicks you're waiting for. If you're interested in double-clicks as well as single clicks, enter the number 2 for clicks. If you're only interested in single clicks, make clicks equal to 1. When a button is first pressed, GEM starts timing a short interval. For each time that the button goes up and down again during that interval, it adds one click to its count. The actual number of clicks is returned in the variable clicked. GEM doesn't check the number of clicks, however, if you're waiting for more than one button.

The length of time during which GEM counts clicks depends on the current double-click speed setting. This setting may be changed from the Control Panel desk accessory that comes with the ST to any of four settings. It's also possible to change this time interval from within a program, using the *evnt_dclick( )* call:

```
int speed_set, speed, flag;
speed_set = evnt_dclick(speed, flag);
```

where *flag* indicates whether you want to read the current double-click setting (flag is 0), or make a new setting (flag is not 0). If you wish to make a new setting, set the speed variable to a number in the range 0–4, where 0 specifies the longest double-click interval, and 4, the shortest. The setting requested, either the current or new setting, is returned in the variable *speed_set*. Since the speed of the double-click setting may affect the speed with which mouse button events are returned, you may wish to change the speed before a mouse button event call and change it back afterwards.

In the *evnt_button()* call, the two input variables *bmask* and *bstate* are used to specify which buttons you want to wait for and whether you want to wait for them to go down or up. The bmask variable is the one you use to specify which buttons to watch. Bit 0 of this word corresponds to the left mouse button, and bit 1, to the right mouse button. If you want to watch the left button, set bmask to 1. If you want to watch the right button you set it to 2. If you want to watch both, set it to 3. The bstate variable is used to specify whether you want to wait for the buttons to go down or up. You set the appropriate bit to 0 if you want to wait for the button to be released, or 1 if you want to wait for it to be pressed.

The mouse button event will only occur when the conditions specified in bstate for the buttons specified in bmask happen at the same time. This means that you can wait for the left button or the right button to go up or down, or you can wait for both. You cannot, however, wait for *either* the left or right button to change state. This makes its impossible for a program to watch for more than one mouse button condition using the standard event-driven input scheme. In order to use both buttons, you will have to return to the polling type of input discussed at the beginning of this chapter. A polling technique will be discussed under the section on the *evnt_multi()* call below.

The actual mouse button state at the end of the double-click interval is returned in the *button* variable. As with the *bstate* variable, the value is 1 if the left button is down, 2 if the right button is down, or 3 if both are down. By checking the mouse button status, the program can determine whether the user merely clicked the button or is holding it down. In addition to the mouse button status, the Shift key status is also reported. The *kstate* variable contains a code that tells whether the right Shift key, the left Shift key, Control key, or Alt key was pressed at the same time as the mouse button. Each of the four low bits represents a different key:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

Thus if *kstate* contains a 4, the Control key was held down when the mouse button was pressed, and if it has a value of 12, both Alt and Control were held. The final two values that *evnt_button* returns are the coordinates of the mouse pointer at the time the button was pressed. Its horizontal position is returned in *mousex,* and the vertical position in *mousey.*

As of this writing, there's a serious bug in the evnt_button( ) routine in the TOS ROMs. If the mouse pointer is moved into the menu bar while your program is waiting for a mouse button event, the machine will lock up. The problem would also occur under similar circumstances if you use the *evnt_multi()* call to await a mouse button event. There's no way to avoid this problem except to use polling techniques (described below) rather than waiting for mouse button events.

Mouse Rectangle Events

When the AES waits for a mouse rectangle event, it watches the mouse pointer's position and informs you when the pointer enters or leaves a designated area on the screen. This elegant system eliminates the need for your program to keep checking the mouse pointer's position. To wait for a mouse rectangle event, you call *evnt_mouse( ),* in the following format:

```
int reserved, mflag, rectx, recty, rectw, recth,
  mousex, mousey, button, kstate;

reserved = evnt_mouse (mflag, rectx, recty, rectw, recth,
  &mousex, &mousey, &button, &kstate);
```

where the *mflag* input variable is used to indicate whether you want to watch for the pointer entering the area or leaving it. A value of 0 means that you want to watch for its entry, and a value of 1 means that you want to watch for its exit. The x position, y position, width, and height of the rectangle are passed in the variables *rectx, recty, rectw,* and *recth.*

As with mouse button events, the status of the mouse buttons is returned in the variable *button,* and the status of the shift keys in the variable *kstate.* The meaning of these coded returns is the same as for the mouse button events, above. GEM reserves a return variable for future use. Currently, a value of 1 is always returned in *reserved.*

Mouse rectangle events have many uses. A common one is to notify the program when the mouse enters a particular area of the screen so the program can change the mouse pointer form. Rectangle events can also be used to detect any movement of the mouse. Simply wait for the pointer to exit from a rectangle one pixel in size and located at the current pointer position

Keyboard Events

You can wait for the user to press a key on the keyboard by calling the function *evnt_keybd( )*. The syntax for this function is

```
int keycode;
keycode = evnt_keybd( );
```

where the variable *keycode* contains a two-byte value that specifies the key or combination of keys struck and the ASCII value of that combination. The first byte usually identifies the key that was struck and isn't affected by shift key combinations. The second byte is the ASCII value of the key combination, which does depend on the state of the shift keys (Shift, Control, and Alt). In most cases, the ASCII value in the low byte is what the program is really looking for, and the high byte can be ignored. Keys which have no ASCII value, however, such as the function keys, return a 0 in the low byte and must be read by the high byte alone. The entire list of keycodes may be found in Appendix B.

Timer Events

You may use the timer event to wait for a specified period of time. The format for the *evnt_timer( )* call is

```
int reserved;
unsigned int timelo, timehi,
   reserved = evnt_timer(timelo, timehi);
```

where *timelo* and *timehi* are the low word and high word of an unsigned 32-bit time period, expressed in milliseconds. This is the opposite of the order the 68000 uses to store a long word, so you'll have to split the 32-bit value into two halves before passing it to this function. In theory, the range of values available allow you to time a period from one millisecond to 65.5

seconds with this function. In practice, however, you'll find that the ST's internal clock does not have sufficient resolution to time an event more closely than to the nearest five or six milliseconds.

A timer event can be used to pace the action of your program or to provide a limit on the amount of time that your program will wait for a particular action. If you specify a time count of 0, the function returns immediately. This "do-nothing" event can be used periodically in your program, if no other AES calls are made, to maintain Screen Manager function. As stated in Chapter 1, the multitasking kernel can only switch tasks when the current one makes an AES call.

Multiple Events

Although each of the event calls was explained separately, there's a way to wait for any or all of these events at once. The *evnt_multi()* call lets you wait for a timer event, keyboard event, message event, mouse button event, and up to two mouse rectangle events, all at the same time. The format for this call is

happened = evnt_multi(events, clicks, bmask, bstate, m1flag, m1rectx, m1recty, m1rectw, m1recth, m2flag, m2rectx, m2recty, m2rectw, m2recth, msgbuf, timelo, timehi, &mousex, &mousey, &button, &kstate, &keycode, &clicked);

Most of these input parameters and output parameters should be familiar from the explanations of the individual event calls. Only the variables *events* and *happened* are new. These are flags which indicate which events you asked to wait for and which events actually occurred. Possible flag values include:

Bit	Bit Value	Macro Name	Event
0	1	MU_KEYBD	Keyboard
1	2	MU_BUTTON	Mouse button
2	4	MU_M1	Mouse rectangle #1
3	8	MU_M2	Mouse rectangle #2
4	16	MU_MESAG	Message
5	32	MU_TIMER	Timer

These can be joined in any combination. For example, a value of 19 in the events flag means that you want the AES to wait for message, mouse button, and keyboard events (16 + 2 + 1). Likewise, the flag for the actual events that occurred is returned in the variable *happened*. Since more than one event can happen at the same time, it's necessary to examine each significant bit of the variable to find all of the events.

The other variables retain the same meanings they had when used in individual event calls. *Clicks, bmask, clicked* and *bstate* are used for mouse button events. The variables that start with *m1* are for the first set of mouse rectangle events, and the ones that start with *m2* are for the second set of mouse rectangle events. *Msgbuf* is used for message events, *keycode* for keyboard events, and *timelo* and *timehi* for timer events. Although *mousex, mousey, button,* and *kstate* are usually associated with mouse button or mouse rectangle events, the values returned in these variables are valid at the conclusion of any *evnt_multi()* call, regardless of whether or not a mouse button or mouse rectangle event occurred.

The GEM literature suggests that you should always use evnt_multi( ) instead of the individual event calls. First, a well-written program usually allows for more than one kind of input from the user. Second, evnt_multi( ) allows you to use the timer in conjunction with other events. This makes it possible to set a time limit on waiting, so that if the user doesn't respond within some reasonable time, your program can recover and help him out, rather than wait forever.

There is a penalty for using evnt_multi( ) from C, however. Because of the large number of input parameters, it takes time to push them all on the stack for the library function, and then it takes the library function time to pull them from the stack, place them in the proper data arrays, make the call, and move the values that were returned from the data arrays back to the output variables.

Since the library routine in the bindings has to be prepared for the unlikely case that you'll want all of the events checked at the same time, you must always supply some value for all of the input parameters, whether you use them or not, and the bindings must move all of those values whether they're used or not. As a result, evnt_multi( ) can be sluggish, particularly when you're closely tracking mouse movement, for instance.

When your application requires faster performance, it may be beneficial to skip the library routines and write your own machine language interface that moves the necessary values to the *int_in* and *addr_in* data arrays before giving the TRAP command that calls the AES.

Polling. As mentioned in the section on mouse button events, in order to use both mouse buttons in a program, you must poll the buttons rather than waiting for button events. You've seen that a timer event of duration 0 returns immediately. So to continuously check the status of the mouse buttons, all you have to do is keep calling evnt_multi( ) in a loop, looking for a timer event of 0. The values for button, kstate, and mousex, and mousey will be updated during each call.

Chapter 4

GEM Graphics Objects

The concept of GEM graphics *objects* is the key to the GEM interface. These objects may be displayed onscreen as plain boxes, boxes with text in them, bit-images, icons, or editable text strings. Anyone who has used the ST is familiar with these objects. For example, the drop-down menus used by GEM programs are composed of GEM objects. The menu bar, the menu titles, and each menu item is a separate GEM object. Each component in a dialog box or an alert box, such as the OK button, which is clicked at the end of an operation, is a GEM graphics object. The icons that appear in the Desktop program, representing disk drives, files, and folders, are all GEM objects. Even the control components in a window, such as the close box, the size box, and move bar are all GEM objects. Simply put, GEM objects are the basic building blocks for all of the sophisticated GEM visual constructs, such as dialog boxes and menus.

At the programming level, these objects are data structures that describe the composition and status of the figures seen on the screen. Most definitions concerning objects can be found in the OBDEFS.H header file. The C definition for an object data structure looks like this:

```
typedef struct object
{
              int   ob_next;    /* object number of next "sibling" */
              int   ob_head;    /* object number of first "child" */
              int   ob_tail;    /* object number of last "child" */
     unsigned int   ob_type;    /* type of object—BOX, CHAR,... */
     unsigned int   ob_flags;   /* flags for color, fill pattern...*/
     unsigned int   ob_state;   /* flags for how to draw—SELECTED,
                                   and so on.*/
              char  *ob_spec;   /* ADDRESS of object-specific info */
              int   ob_x;       /* left edge of object */
              int   ob_y;       /* top edge of object */
              int   ob_width;   /* width of object */
              int   ob_height;  /* height of object */
} OBJECT;
```

Since objects are so important to GEM, and since the composition of object data structures is somewhat complex, each member of the data structure will be explained in detail below.

Object Tree Structure

While the concept of individual objects is a powerful one, the way GEM combines objects makes them even more powerful. A menu bar, for example, is made up of a number of individual menu title objects and menu item objects. But in order for these objects to function together to make up a menu, it's necessary to define a relationship between them. The first three fields in the data structure for each object are used as pointers to other objects. These pointers, *ob_next, ob_head,* and *ob_tail,* define the relationship between objects, and make it possible, for example, to draw a series of related objects with a single command.

The system used to link objects is called a *binary tree.* If you look at a diagram of such a tree, you'll notice that it resembles a family tree. A family tree starts with the earliest known common ancestor at the top, and shows each generation branching out downward, so that the tree grows larger and larger the farther down you go. A binary tree works much the same way, and even uses the same terminology, so that *parent* objects may spawn groups of related objects, which are referred to as their *children.*

A *struct* is a data structure as used in the C programming language. Structs might be formed of any kind of data. The type of struct is determined by the data it contains. Thus, if the struct contains object data, it is a struct of the object type. An object tree is simply an array of object structs.

The object number of an object is the array index number of that object. The root object (object number 0) sits at the top of each object tree. The screen display rectangle of the root object completely encloses all of the other objects in the tree. The largest objects within the root are the sibling objects which are all children of the root object. Each of these objects may contain their own, smaller child objects.

The root object, like all other objects, contains three linkage fields at the beginning of its data structure. These fields are called *ob_next, ob_head,* and *ob_tail.* The NEXT field contains the object number of the next object which is a child of the same parent as the current object (such objects are known as *siblings*). Since the root object by definition has no parent, it can have no siblings. Therefore, its NEXT field contains a −1, which means that there's no link in this direction. For objects which do have siblings, the NEXT field of each contains the object number of the next sibling. The last sibling at a particular level contains the object number of the parent in its NEXT field. If an object has children (the root object nearly always has children), the HEAD link field contains the object number of the first child, and the TAIL field contains the object number of the last child.

To better understand how object trees work, it would be useful to examine one. Figure 4-1 shows a sample GEM dialog box. This box is made up of seven objects. The root object is the G_BOX that surrounds all of the other objects, which we'll call DIALBOX. Object 1, OKBUTTON, is the box labeled OK. Its sibling, Object 2, PATBOX, is the pattern-filled box that holds the rest of the objects. Its two children are Object 7, INSTRUCT (the SELECT...text box) and Object 3, a box named INVISBOX. Don't strain your eyes looking for INVISBOX—as it's name suggests, it's invisible. This box surrounds its three children, the box objects marked *A*, *B*, and *C*. These are Objects 4, 5, and 6, which we'll call BUTNA, BUTNB, and BUTNC, respectively.

Figure 4-1. GEM Dialog Box and Its Contents

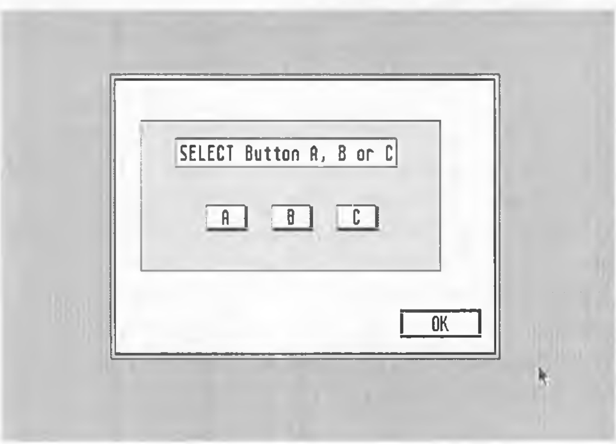
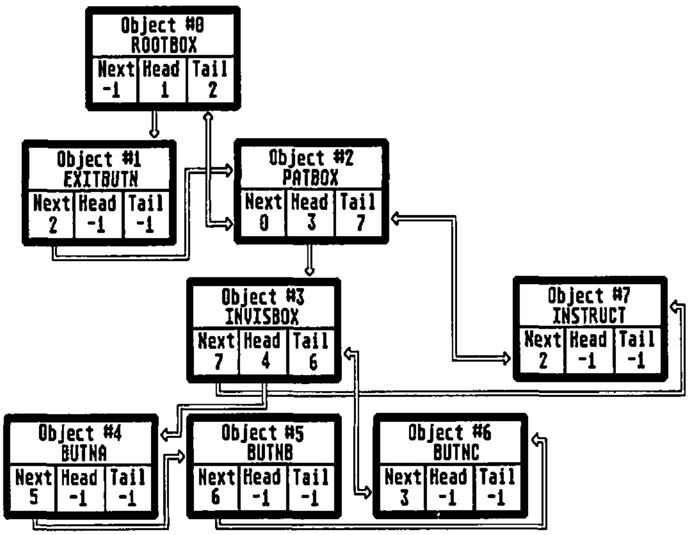


Figure 4-2 illustrates the link fields for the seven objects. Object 0, the root, has no siblings in its NEXT field, but its HEAD field points to its first child, object 1, and its TAIL field points to its second (and last) child, object 2. Object 1 has no children, so HEAD and TAIL contain a −1, but it does have a sibling, so NEXT points to object 2. Since object 2 has no more siblings, its NEXT field points back to object 0. Its HEAD field points to object 3, its first child, and its TAIL field points to object 7, its last child.

Figure 4-2. Link Fields for Objects in Dialog Box



If you look at the sample object tree, you may find the numbering of objects a bit odd, with objects 3 and 7 at one level, and objects 4, 5, and 6 at a lower level. This is the order in which these objects are written out by the Resource Construction program which we'll discuss in the next chapter, and it's indicative of the order used to walk the tree. Walking the tree is a process by which you start with the root object and visit each of the subordinate objects in turn. Using the link fields, you can examine each of the objects in the tree by using the following procedure:

1. Examine the *ob_head* field of the current object. If there's a child, visit it by making it the current object. Then go back to Step 1.
2. If there's no child, look at the *ob_next* field. If this field contains a NIL (−1) value, you're back at the root object, and your tour is finished.
3. If the value in ob_next is not NIL, check to see if the object it points to contains the number of the current object in its

ob_tail field. If it does, we've moved back up to the parent of the current object, so make it the new current object and go back to Step 2.

4. If the value in ob_next is not NIL, and does not point to an object whose ob_tail is the same as the current object, then it points to a sibling object which you haven't visited. Visit this object by making it the current one, and go back to Step 1.

The AES performs many of its tasks by following the object tree. For example, it can draw the entire tree by starting at the root and then drawing each object it visits as it walks around the tree. Your own program can also use these links to perform a specified operation on all of the members of a tree, such as changing all of their status flags.

Although the scheme used to link GEM objects together may seem complex, it actually simplifies many tasks which the AES must perform. To understand how, you must look at the last four fields of the object structure.

Object Size and Position

The final four fields of the object structure contain the position and size of the object, expressed in the standard format of *x* position, *y* position, width, and height. In this case, however, the *x* and *y* positions don't represent an absolute pixel coordinate. Rather, the position is specified as an offset from the *x,y* position of its parent object (whose own *x,y* position is an offset from its parent's position). Thus, each object inherits the origin position of its parent. Changing the position of a parent object also changes the positions of all of its descendants.

The relative positioning of objects within their parent's rectangle is just one aspect of the principle known as the *visual hierarchy* of GEM objects. This principle states that the rectangle of each parent object completely contains the rectangles of all of its children. Since each parent rectangle contains all of its child rectangles, it stands to reason that when the parent object moves, so must the children.

One of the most important tasks which the AES must perform in order to maintain the system of GEM objects is finding which object rectangles (if any) overlap a given *x,y* location on the screen. The hierarchy of GEM objects greatly simplifies this task, by breaking the screen down into a number of significant rectangles. The largest of these is the rectangle of the root object. If the location doesn't lie within the root rectangle, it can't coincide with any object in that tree. If it's within the root, the AES then checks it against the rectangles of the first level of siblings. If it lies within one of those rectangles, the AES keeps moving down the tree until it finds the smallest object whose rectangle contains the point in question.

But if the point doesn't lie within the rectangle of a particular parent object, the AES can forego checking any of its children, since none of the children can contain a point that isn't located within the parent. You should note that while the object tree hierarchy insures that a child will be selected over its parent, it doesn't prevent two siblings from overlapping. If they do, and the user clicks in the overlap area, the AES will find the last or "rightmost" sibling (according to the tree diagram). That's why you can use an I_BOX sibling to cover other siblings so that all will appear to be selected at once (as you'll see later in the section on box-type objects).

The tree structure also helps the AES in drawing object trees. The visual hierarchy insures that smaller objects are always drawn after larger objects, so the larger parent won't cover up its smaller children. As you'll see later, the *objc_draw()* function that draws an object tree allows you to specify a clipping rectangle within which the objects will be drawn. Because of the tree structure, this drawing routine can automatically skip drawing any child object if its parent is not within the clipping rectangle.

The usefulness of the mouse rectangle event becomes much more apparent when considered in the light of the object tree structure. Using parent objects to contain subobjects allows you to divide the screen into large rectangles which can be used as mouse rectangles. These can be further subdivided into large areas containing several child objects each. Once the user has clicked in the largest rectangle, you can follow the object tree into smaller and smaller rectangles, eliminating on the way all objects that are not within the parent rectangles. This strategy lies at the heart of the point-and-click operation of GEM.

Types of GEM Objects

The *ob_type* field in the object data structure is used to describe the type of object. The standard GEM object types, and their macro names (from the OBDEFS.H file), are as follows:

Type Number	Macro Name	Description
20	G_BOX	An opaque box (with optional border)
21	G_TEXT	A formatted text string for which you may specify color, font size, and horizontal positioning
22	G_BOXTEXT	A formatted text string within an opaque box (with optional border)
23	G_IMAGE	A monochrome graphic bit-image
24	G_PROGDEF	A user-defined object type
25	G_IBOX	A transparent box (with optional border)
26	G_BUTTON	A G_STRING (see below) with a border around the text
27	G_BOXCHAR	A single text character within an opaque box (with optional border)
28	G_STRING	A string of black graphics text, in the default font
29	G_FTEXT	An editable text field, for which you may specify a template, a validation string, and an initial value.
30	G_FBOXTEXT	An editable text field within an opaque box (with optional border)
31	G_ICON	An object consisting of a monochrome bit-image and an image mask used for for drawing the image in normal and inverse video modes
32	G_TITLE	A specially formatted G_STRING for use in the title bar of menus

As you can see, the various objects types are mainly made up of combinations of rectangular boxes, text forms, and bit-images. Each of the object types will be examined in detail below. At the same time, there will be an examination of the *ob_spec* data field which contains either two words of object-specific information, or a pointer to an object-specific information block. A point which may be of interest to you is that the AES doesn't use the top byte of the object type field. Programmers are free to use this byte to create their own extended object types, which contain additional information about the object.

Box Objects

The three box object types are G_BOX, B_IBOX, and B_BOXCHAR.

G_BOX. The first, G_BOX, is a solid rectangle with an optional border. The box may be filled either with a solid color or a pattern. A G_BOX is generally used as a backdrop on which to place other objects. The dialog box illustrated in Figure 4-1, for example, uses a G_BOX for the root object ROOTBOX, the plain white box on which the entire dialog is mounted. It also uses a patterned G_BOX (PATBOX) as a patterned backdrop for the three buttons and the text.

G_IBOX. The second type of box object, G_IBOX, is a transparent box with an optional border. If the thickness of the border is set to 0, the box is truly invisible, as its name suggests. A G_IBOX is most often used to group together other objects. For example, in the dialog box of Figure 4-1, the three radio buttons marked A, B, and C are contained in an invisible G_IBOX called INVISBOX. Since radio buttons must be siblings (as you'll see later on), the G_IBOX is used as an invisible common parent. Another possible use for a G_IBOX is to group together two or more objects in order to make any of them select all of them. For example, say that two sibling objects—one an image and the other a text string—are placed near each other. If you cover them with a higher-numbered sibling G_IBOX, both will still show through, since the G_IBOX is transparent. But clicking anywhere in the box will highlight both the image and the string.

G_BOXCHAR. The final box object, G_BOXCHAR, is an opaque box like B_BOX, only with a single text character drawn in the center of the box. The G_BOXCHAR type of object is used by GEM for the various window controls, such as the sizer, fuller, and closer. In the example dialog, G_BOXCHARs were used for the three radio buttons, BUTNA, BUTNB, and BUTNC.

The *ob_spec* field for each of these box type of objects contains two words of data that describe the color of the object, and the thickness of its borders. The low word is used for the object color:

Bit Numbers	Contents	
0–3	Interior color (0–15)	
4–6	Interior fill pattern	
	0	Background color (IP_HOLLOW)
	1–6	Dither patterns of increasing darkness (IP_1PATT to IP_6PATT)
	7	Foreground color (IP_SOLID)
7	Writing mode	
	0	Transparent
	1	Replace
8–11	Border Color (0–15)	
12–15	Text Color (0–15)	

The names in parentheses next to the interior fill pattern codes are the macro names given for these patterns in OBDEFS.H. The default colors and their macro names are listed below:

Color Number	Name
0	White
1	Black
2	Red
3	Green
4	Blue
5	Cyan
6	Yellow
7	Magenta
8	Low white (light gray)
9	Light black (dark gray)
10	Light red
11	Light green
12	Light blue
13	Light cyan
14	Light yellow
15	Light magenta

These colors are only the default values. The color contained in any particular color register can be changed by the user at any time from the Control Panel desk accessory. These colors may also be changed by a program that uses the *vs_color()* command. It's worth noting that while any color number in the range 0–15 is valid, medium resolution screens can only display the first four colors, while monochrome screens are limited to the first two. Any out-of-range selections will be treated as color 1 (black) on these displays.

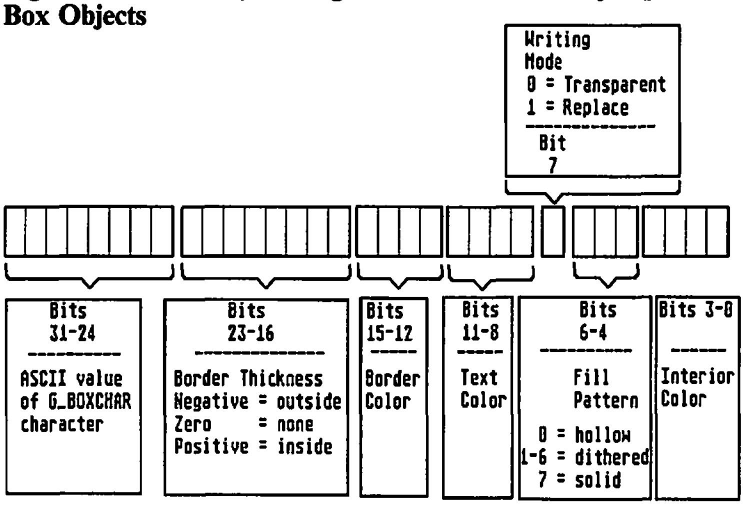
The low byte of the high word of the ob_spec field of box objects is used for the thickness of the border:

−127 to −1	Outside thickness in pixels outward from object's edge
0	No thickness
1 to 128	Inside thickness in pixels inward from object's edge

Since the VDI drawing routines used to render wide lines only respond to odd-numbered settings, the border thickness value that you specify should be an odd number.

The high byte of the high word is used only for the G_BOXCHAR object type. It contains the ASCII value of the character drawn in the box. The chart in Figure 4-3 summarizes the flags stored in the 32-bit objc_spec field of box objects.

Figure 4-3. Summary of Flags Stored in 32-Bit Objc-Spec Field of Box Objects



Text String Objects

The next group of objects, G_STRING, G_BUTTON, and G_TITLE, contain text strings. All three of these object types use the plainest text, drawn in the default size of the system font, with black as the text color, and with no special effects such as italics.

The ob_spec field of all of the string objects contains a pointer to the string. This pointer is the long-word address of the first character of the text. The string must follow the C language convention of ending in the ASCII 0 character.

G_STRING. The G_STRING type is used for fixed, explanatory text.

G_BUTTON. The G_BUTTON is one of the most frequently used types of objects. It's like a G_STRING with a one-pixel border drawn around the text. As you will see below, the width of this border is increased if certain attribute flags such as DEFAULT are set. Buttons are most often used as controls the user can click on. For example, the OKBUTTON object in the sample tree was a G_BUTTON type of object that the user could click on to end the dialog. In addition to buttons that exit the dialog, like OK and CANCEL, these object types can be used as Boolean on-off switches, allowing the user to select an option or to deselect it.

G_TITLE. The last type of string object is the G_TITLE. This is a string object which is specially formatted for use in the title bar of menus. Its special formatting insures that the menus are redrawn correctly.

Formatted Text Objects

The next group of text objects, G_TEXT, G_BOXTEXT, G_FTEXT, and G_FBOXTEXT, are a little bit fancier than the previously mentioned string objects. All of these objects allow you to select normal or small text, different text colors, and horizontally justified type. In addition, two of them allow you to specify ways the user will be able to edit the text string.

G_TEXT and G_BOXTEXT. The first two, G_TEXT and G_BOXTEXT, are just fancier versions of the G_STRING and G_BUTTON types. G_TEXT is a colored text string, while B_BOXTEXT is a formatted text string surrounded by a border. These object types can be used in place of the simpler string objects when you want to use a text color other than black, a smaller text font size, or if you need horizontal positioning to take place when the program executes. Since they require more memory than the G_STRING and B_BUTTON types and, because too much color can distract the user, you should exercise discretion in the use of these object types.

G_FTEXT and G_FBOXTEXT. G_FTEXT and G_FBOXTEXT are editable versions of G_TEXT and G_BOXTEXT. This doesn't mean that they're inherently editable. Rather it means that you supply enough information about them to allow Library routines like *form_do( ),* which handles all dialog activity, to supervise editing by the user. The editing information you supply consists of a template which determines the text format, a validation string that shows which characters may be entered in which positions, and an initial or default value for the string.

Editing and display information is stored in a data structure called a TEDINFO. The ob_spec field of text type objects contains a pointer to the object's TEDINFO. The C language definition of this data structure is as follows:

```
typedef struct text_edinfo
{
   char  *te_ptext;       /* pointer to the actual text string */
   char  *te_ptmplt;      /* pointer to format template */
   char  *te_pvalid;      /* pointer to validation string */
   int    te_font;        /* font size (3 = normal, 5 = small)*/
   int    te_resvd1;      /* reserved word */
   int    te_just;        /* horizontal justification—left, right... */
   int    te_color;       /* color information word */
   int    te_resvd2;      /* reserved word */
   int    te_thickness;   /* border thickness */
   int    te_txtlen;      /* length of text string */
   int    te_tmplen;      /* length of template string */
} TEDINFO;
```

The first three fields are four-byte pointers to text strings. The first, *te_ptext,* is a pointer to the actual text string. This string may be composed of either text characters or blank spaces. Note that the commercial at sign (@) is treated as a special case. If it appears as the first character in the string, the entire string is considered to be composed of spaces. While this is handy for the programmer, if the user edits the first character of the string to this symbol, the string will be blank the next time the text object is displayed.

The next field, *te_ptmplt,* is a pointer to the template string. This string controls the format in which the text field is displayed. It is composed of constant text characters that won't be edited and underscore characters which indicate the position of at which the editable text is entered. For example, say that the editable field is being used for the entry of a telephone number. The template string might be "Phone Number:(___)___-____". If the initial string was "1234567890", the text object would be printed as "Phone Number:(123)456-7890". If the initial string did not have enough characters to fill out the template string, the unfilled underscore characters would be printed.

The third field, *te_pvalid,* is a pointer to the validation string. This is a string that shows which type of character may be entered at a given character position. This string is composed of special validation characters, each of which stands for a class of allowable input:

Validation Character	Input Accepted
9	Numeric digits 0–9 only
a	Upper- and lowercase alphabetic characters and the space character
n	Numeric digits 0–9, upper- and lowercase alphabetic characters, and the space character
p	All valid DOS pathname characters, including the colon (:) and backslash (\ )
A	Uppercase alphabetic characters and the space character
N	Numeric digits 0–9, uppercase alphabetic characters, and the space character
F	All valid DOS filename characters, including the colon (:), and wildcard characters question mark (?) and asterisk (*)
P	All valid DOS pathname characters, including the colon (:), the backslash (\ ), and wildcard characters question mark (?) and asterisk (*)
X	Any character

In the telephone-number example, you would use a validation string of 9999999999, since you only want to allow the user to enter a number in the blanks.

The next field in the TEDINFO structure is *te_font.* This field allows you to select the normal-sized system font that's used to draw menu items or the small-sized font that's used to draw the text under the icons on the Desktop. A value of 3 selects the normal text font, while a value of 5 selects the small text font. The next field, *te_resvd1,* is reserved for future use. After that comes *te_just,* which contains a flag indicating the type of horizontal text justification used. A value of 0 selects left-justified text, a value of 1 selects text that is right-justified. A value of 2 indicates that the text should be centered.

The *te_color* field is a word of bit flags that identifies the color and pattern used for box type objects. Its format is exactly the same as the low word of the ob_spec field for the box objects, described above. The *te_thickness* field is used to describe the thickness of the border surrounding the G_BOXTEXT or G_FBOXTEXT. Its format is the same as that of the low byte of the high word of the ob_spec for box object (negative numbers to −127 show the outside thickness in pixels, positive numbers to 128 show the inside thickness in pixels).

The last two fields in the TEDINFO structure are *te_txtlen* and *te_tmplen*. These fields specify the length of the text pointed to by te_ptext, and the length of the template string pointed to by *te_ptmplt*. These string sizes should include the null character (ASCII 0) used to terminate each string.

Bit-Image Objects and Icons

The next object type, G_IMAGE, is a simple monochrome bit-image object. This type of object is usually used to display a picture that's not meant to be selected by clicking the mouse on it. The reason these objects usually aren't selectable is because only a single bit-plane image is supplied, so selecting them causes the whole rectangle around the image to be inverted. Examples of bit-image objects are the drawings used to illustrate the Note, Wait, and Stop alert messages.

The *ob-spec* field of a G_IMAGE contains a pointer to a data structure called a BITBLK. The C language definition of this data structure is

```
typedef struct bit_block
{
  int  *bi_pdata;  /* pointer to bit-image data array */
  int   bi_wb;     /* width of bit image in bytes */
  int   bi_hl;     /* height of bit image in scan lines */
  int   bi_x;      /* Image X offset from start of data block */
  int   bi_y;      /* Image Y offset from start of data block */
  int   bi_color;  /* foreground color of image */
} BITBLK;
```

The *bi_pdata* field contains a pointer to the bit-image data array. This array must be composed of 16-bit words. Therefore, the image's width in pixels must be an even multiple of 16. The bit-image data uses a simple coding scheme, where each bit represents a pixel set to the foreground color (1) or the background color (0). The next field, *bi_wb*, gives the width of the bit image in bytes. Since the data array is composed of words, this field must always contain an even number. The *bi_hl* field contains the height of the bit image in scan lines. The fields *bi_x* and *bi_y* allow you to specify an *x* and *y* offset from the beginning of the image data array. This allows you to use only a portion of the image data for the object's bit image. The last field, *bi_color*, allows you to select the color used to draw pixels that represent the bits that are set to 1. Although the image itself is all the same color, this color need not be restricted to black. Any color register from 0 to 15 is valid, although in medium and high resolutions, out-of-range numbers will be converted to 1 (black).

There is another, more sophisticated type of bit-image object called a G_ICON. This type of object not only uses a bit-image data array, but an image mask as well. This image mask allows only the shape of the image to be highlighted when the object is selected. The G_ICON image may also contain a bit of explanatory text or an attached title, as well as a single text character superimposed somewhere on the image. Finally, it may specify both the color of foreground and background data bits. Examples of icon objects are easy to spot in the GEM Desktop program—all of the disk drive, trash can, file folder, and file image are G_ICONS.

The ob-spec field of a G_ICON type object contains a pointer to a data array called an ICONBLK. The C language definition for this data structure is

```
typedef struct icon_block
{
  int   *ib_pmask;  /* pointer to the image mask data array */
  int   *ib_pdata;  /* pointer to the bit-image data array */
  char  *ib_ptext;  /* pointer to the object's text string */
  int    ib_char;   /* low byte = ASCII character drawn on icon */
                    /* high byte = foreground and background colors */
  int    ib_xchar;  /* the X offset of that character */
  int    ib_ychar;  /* the Y offset of the character */
  int    ib_xicon;  /* the X coordinate of the icon */
  int    ib_yicon;  /* the Y coordinate of the icon */
  int    ib_wicon;  /* the width of the icon in pixels */
```

```
int   ib_hicon;   /* the height of the icon in pixels */
int   ib_xtext;   /* the X offset of the text string */
int   ib_ytext;   /* the Y offset of the text string */
int   ib_wtext;   /* the width of the text in pixels */
int   ib_htext;   /* the height of the text in pixels */
} ICONBLK;
```

In order for you to understand the contents of the ICONBLK structure, you should first understand how the AES draws G_ICON images. An icon is composed of two bit images, the mask and data images. The *ib_pmask* field of the ICONBLK contains a pointer to the mask data array, while the *ib_pdata* field contains a pointer to the bit-image data array. When the AES draws the icon, it first draws the image mask, setting all of the pixels that correspond to 1 bits in the mask to the background color. It then draws in the data image, setting all of the pixels that correspond to 1 bits in the data to the foreground color. When the icon is selected, the AES highlights the image by drawing the mask in the foreground color and the data image in the background color. The field *ib_wicon* contains the width of the icon image in pixels. Since the mask and image data arrays are composed of 16-bit words, this width must be an even multiple of 16. The *ib_hicon* field contains the height of the icon in pixels. The *ib_xicon* and *ib_yicon* fields contain the *x* and *y* coordinates of the icon image, relative to the data arrays.

As mentioned above, you may specify a text string to appear with the icon image, such as the filename that appears under the file icons on the Desktop. This text string is printed using the small size of the system text font. The *ib_ptext* field contains a pointer to the null-terminated text string itself. The *ib_xtext* and *ib_ytext* contain the *x* and *y* offsets for the start of the text, relative to the upper left corner of the icon itself. And the *ib_wtext* and *ib_htext* fields contain the width and height, respectively, of the text string in pixels.

You may also specify a text character to appear with the icon image, such as the drive numbers that appear on the file drawer icons on the Desktop. The low byte of the *ib_char* field contains the ASCII value of the text character to appear with the icon. The *ib_xchar* and *ib_ychar* fields contain the *x* and *y* offsets for the character, relative to the top left corner of the icon.

Although not noted in the GEM documentation, the high byte of the *ib_character* field contains the foreground and background color indices. The upper four bits select the foreground color, while the lower four bits select the background color. While it's possible to select any combination of foreground and background colors, in order to retain compatibility across display modes and in the interests of good taste, you'll almost always want to stick with black for the foreground color and white for the background.

User-Defined Objects

The final object type is the most flexible of all. It is the user-defined object type, G_PROGDEF (in some versions of the OBDEFS.H file, the macro name for this type may be G_USERDEF). This type allows the programmer to create object types with an appearance different from the standard box, string, or image objects. For example, you might create an object type that consists of a box with rounded corners or a circle.

The *ob_spec* field of a G_PROGDEF object contains a pointer to a data structure called an APPLBLK (in some versions of OBDEFS.H it may be called a USERBLK). The C language definition of this data structure is

```
typedef struct appl_blk
{
  long (*ub_code)( );
  long ub_parm;
} APPLBLK;
```

As you can see, it consists of two long word values. The first is a pointer to the function supplied by the user for drawing the object. The second is a user-defined parameter that's passed to the routine when the drawing takes place. This parameter is passed as part of a larger block of information called a PARMBLK structure.

The AES calls the user-defined drawing function whenever the G_USERDEF needs to be drawn or modified. This occurs when the *objc_draw()* routine is used to draw the object, or the *objc_change()* routine is used to change its state. Say, for example, the *ub_code* field of an object's APPLBLK points to the function *user_draw()*. Whenever that object is due to be drawn, the AES calls user_draw( ), and passes it a pointer to a PARMBLK structure. The C language definition of this structure is

```
typedef struct parm_blk
{
  OBJECT *pb_tree;       /* Pointer to the object tree */
                         /* containing this object */
  int    pb_objc;        /* This object's index in the tree */
  int    pb_prevstate;   /* The object's previous (old) state field */
  int    pb_currstate;   /* The object's current (new) state field */
  int    pb_x,           /* The position and size of the object */
         pb_y,
         pb_w,
         pb_h;
  int    pb_xc,          /* The position and size of the clip rectangle */
         pb_yc,
         pb_wc,
         pb_hc;
  long   pb_parm;        /* The contents of the ub_parm field */
                         /* in the APPLBLK */
} PARMBLK;
```

This structure gives the drawing function much of the information needed to draw the object. The *pb_tree* field contains a pointer to the object tree that contains this object, and the *pb_objc* contains the object number of that object within the tree. The *pb_prevstate* and *pb_currstate* fields contain the previous and current values for the object's state flag (the possible state values will be discussed below). If the previous and current states are the same, the application is drawing the object. If they are different, the application is changing the object's state.

The next eight words contain rectangle information for the object and for the current clipping rectangle. The object's *x* and *y* positions given here represent the actual pixel position of the object, not an offset from its parent. The clipping rectangle, likewise, represents a rectangle on the physical screen. The last field in the PARMBLK structure is called *pb_parm,* and it contains whatever value you placed in the *up_parm* field of the APPLBLK structure.

To summarize, when you define a G_PROGDEF type object, you place a pointer to an APPLBK structure in its ob_spec field. That APPLBLK contains the address of a drawing routine and a long word parameter. When the object must be drawn, or when its state field changes, the AES calls your drawing routine. The AES passes a pointer to your drawing routine which points to a PARMBLK, containing information about the object's size, position, and state. For example, if your APPLBLK points to a drawing routine called *user_code()*. This routine would be defined like this:

```
user_code(pb)
        PARMBLK *pb;
{
  read_parms( )  /* read whatever parameters necessary from
                    PARMBLK */
  do_draw( )     /* draw the object, or whatever else you want to
                    do */
  return(0)      /* your routine should always return a value of
                    zero */
}
```

A few words of caution are in order about what your drawing code can contain. First, your routine is being called by the AES and uses the AES's stack. Therefore, avoid programming techniques which use a large amount of stack space. For instance, you will probably want to avoid using long parameter lists or recursion.

Second, the AES is not reentrant, and so you may not call AES routines from within your drawing code (VDI routines are all right).

Finally, your routine should always return a value of 0 to the AES. This signals that everything's all right. If any other value is returned, the AES drawing operation will halt, and the other objects in the tree won't be drawn.

Object Flags

The next data field in the object structure is called *ob_flags*. This field is used to specify attributes for the object which indicate how the user should be able to interact with it. This interaction is usually carried out by means of AES Library routines such as *form_do()*, which controls all of the user interaction during dialogs. This routine examines the various object flags and implements their functions.

The object attributes are set up as bit flags, which means that each is assigned a bit in the word ob_flags. If the bit is set to 1, the object has the attribute, and if it's reset to 0, the object doesn't. The full list of bit flags and the macro name assigned them in the OBDEFS.H header file is as follows:

Bit Number	Bit Value	Macro Name	Description of Attribute
0	1	SELECTABLE	The user can select the object by clicking on it. A change in graphics, such as color inversion, should indicate the selected state.
1	2	DEFAULT	The user can select the object from a dialog by pressing Return key. There must only be only one object with this flag per tree, and some graphic change, such as thickening of the button border, should indicate that this object is the default.
2	4	EXIT	Selecting this object indicates that the user wishes to exit the dialog.
3	8	EDITABLE	Indicates the user is able to edit the object in some way (such as changing the text string).
4	16	RBUTTON	This object is one of a set of radio buttons. Radio buttons are sibling objects whose selection is mutually exclusive—selecting one automatically deselects the others.
5	32	LASTOB	The object is the last one in the object tree.
6	64	TOUCHEXIT	Indicates that the form_do( ) routine will return control to the program as soon as the user presses the mouse button with the mouse touching the object, without waiting for the button to be released as with EXIT.
7	128	HIDETREE	Makes a subtree invisible. Object library calls will not draw or find the object or any of its children.
8	256	INDIRECT	This flag to show that the value in ob_spec is actually a pointer to the real value of ob_spec.

The SELECTABLE flag. The SELECTABLE flag is an important one. If this flag is set, the user can select the object by moving the mouse pointer over it and clicking the left mouse button. When this happens, the AES changes the flag which indicated whether the item is selected (see the SELECTED flag, below) and redraws the entire object rectangle in reverse video to highlight the object. The select mechanism works as a toggle. If the object is selected and the user clicks on it, it is deselected. While the AES takes care of the details of updating the object state flag and the video display, it is up to the application to check the SELECTED flag and, if it is set, perform the specified task.

The DEFAULT flag. A related flag is the DEFAULT flag. The form_do( ) library function uses this flag in handling keyboard events during a dialog. If the Return key is pressed, the default object is automatically selected, and the dialog is concluded. Note that the default object may be selected even if the SELECTABLE flag is not set or if the DISABLED state flag is set, and that selecting the default object with the Return key causes the dialog to exit whether or not the EXIT flag is set for this object. The AES makes some default objects (such as buttons or boxes with an outside border) visually distinct by thickening the border surrounding them by one pixel. For that reason, you should only set the DEFAULT flag for one object in a dialog, since the AES will only select the first default object it finds, and you don't want to mislead the user into thinking that another one is the default. In a dialog, you'll often see a button such as OK or CANCEL with a thick box around it, indicating that it's the default object.

The EXIT flag. The EXIT flag indicates that the AES Library routine form_do( ) will exit a dialog when this object is selected by moving the mouse pointer over it and clicking the left mouse button (the SELECTABLE flag must also be set). As with default objects, the AES makes some exit objects visually distinct by thickening the border surrounding them by one pixel. If both flags are set, the border around the default object is thickened by two pixels. You'll often see a dialog with buttons like OK and CANCEL where the borders of both are thickened, but that of the default button is the thicker of the two.

The EDITABLE flag. The EDITABLE flag is used to indicate that an object may be edited by user interaction. The form_do( ) library routine in particular uses this to locate editable text objects. This helps the form_do( ) keyboard handler routine to find the next editable text field when the user presses the Tab or Shift-Tab key combinations.

The RBUTTON flag. Sometimes in a dialog, you wish to present a number of options which are mutually exclusive. For example, in a communications terminal program, you might want to allow the user to set the data transfer speed to 300, 1200, or 2400 bits per second. Therefore, it's handy to be able to indicate that the act of selecting some objects will automatically deselect others. The RBUTTON flag is used for just this purpose. It indicates that the object is a radio button, named for the push buttons on a car radio which pop up when a new one is pushed. In order to indicate which other objects share this mutual exclude feature, radio buttons must be siblings. Radio button objects are normally set up by creating a box object, such as a G_BOX or G_IBOX, and placing the buttons inside. The sample dialog box of Figure 4-1 is arranged this way, with BUTNA, BUTNB, and BUTNC all children of the invisible INVISBOX.

The LASTOB flag. The LASTOB flag merely indicates that an object is the last one in its object tree.

The TOUCHEXIT flag. The TOUCHEXIT flag is an interesting variation on the normal EXIT. When this bit is set in an object's ob_flag field, form_do( ) exits as soon as you move the mouse over the object and press the left mouse button. It does not wait until the mouse button is let up again, as it does for exit objects. This allows you to create your own scroll bars or other dragable or autorepeat controls within a dialog. While the AES doesn't wait for the button to be released on a TOUCHEXIT object, it does test to see if it has been released. If it finds that the button was released, it waits to see if the user double-clicks. When it detects a double-click, it sets the high bit of the object number returned from form_do( ). Therefore, your program code should be prepared to deal with the possibility that the user will double-click on a TOUCHEXIT object. At the very least, it should mask off the top bit of the object number returned by form_do( ) if it doesn't care about double-clicks. Of course, you can always use the double-click condition to enhance the function of the object, like allowing the user to double-click on a filename to open the file immediately.

The HIDETREE flag. The HIDETREE flag is used to make part of an object tree invisible. Whenever the *objc_draw()* or *objc_find()* routines are called, the AES will not draw or find this object or any of its descendants.

The INDIRECT flag. Finally, the INDIRECT flag is used to change the meaning of the ob_spec field. When INDIRECT is set, ob_spec is interpreted as a pointer to the actual ob_spec data. This allows you make the actual ob_spec part of a larger block of user-defined data, with the new ob_spec a pointer to that data block.

Object States

The final field in the object structure is called *ob_states*. The state of an object determines what the application does with it, as well as the way it's drawn by the library routine *objc_draw()*. This field, like the flags field, is made up of bit flags. The various flags and their macro names (from the OBDEFS.H header file) are listed below:

Bit Number	Bit Value	Macro Name	Description
0	1	SELECTED	The object is highlighted, usually by inverting the foreground and background colors.
1	2	CROSSED	For a box-type object, indicates that an X is drawn in the box.
2	4	CHECKED	Indicates that a text object such as a menu title is drawn with a checkmark in front of it.
3	8	DISABLED	Indicates that the object (typically a menu title) is drawn faintly by masking out every other pixel.
4	16	OUTLINED	An outline is drawn around a box object.
5	32	SHADOWED	The object (usually a box) is drawn with a drop shadow.

The SELECTED flag. The SELECTED flag indicates that the user has selected this object, usually by moving the mouse pointer over it and clicking the left button during the course of a dialog. When the objc_state( ) function is used to change an object's state to selected, the AES visually highlights the object, usually by inverting the foreground and background colors in its rectangle. Icons are the exception to this rule, in that their image and mask colors are inverted, not the colors of their entire rectangle.

The CROSSED and CHECKED flags. The CROSSED and CHECKED flags are generally used to manage an alternate system of highlighting a selected object. CROSSED is used with filled box objects to draw an X in the box using the background color. By setting the TOUCHEXIT flag, the application can easily toggle the CROSSED flag, turning the X on or off.

The CHECKED flag is used with text boxes, such as those found in menu items. It places a checkmark in front of the text. The Menu Library routines include *menu_icheck()* which is used to display or erase the checkmark in front of menu items.

The DISABLED flag. The DISABLED flag is used to indicate that an object (usually a menu item or button) may not be selected. The AES draws a DISABLED object faintly, by superimposing a crosshatch of white dots over it. The form_do( ) routine will not allow the user to select a DISABLED object by clicking on it, though it will allow the selection of a DEFAULT DISABLED object by pressing Return.

The OUTLINED and SHADOWED flags. The final two states, OUTLINED and SHADOWED are used mostly for cosmetic enhancement of container-type box objects. The OUTLINED state causes the AES to draw the object with a thin outline around it. Combined with a two- or three-pixel inner border, this gives the object an attractive frame.

The SHADOWED state causes the AES to draw a *drop shadow* under the object. This is a two-pixel line drawn underneath and to the right of the object. Root objects of dialog boxes often have a drop shadow under them.

Object Library Routines

The AES provides a number of low-level routines for dealing with objects. The most basic routine draws an object tree. You'll recall that an object tree is a linked array of object data structures. It's called *objc_draw()*. Here is how it is called:

```
int status, firstob, depth, clipx, clipy, clipw, cliph;
struct object tree[ ];

status = objc_draw(tree, firstob, depth, clipx, clipy, clipw, cliph);
```

This function draws the object tree whose address is contained in the pointer tree, starting with the object whose index number is in *firstob* and going down as many generations as specified in the *depth* variable. If you want the whole tree drawn, you may use ROOT (defined as 0) for firstob and MAX_DEPTH (defined as 8 in OBDEFS.H) as the depth variable. This function also allows you to specify a clipping rectangle, whose position and size are given in *clipx, clipy, clipw,* and *cliph.* Only those portions of objects which are contained within the clipping rectangle will be drawn. As explained in connection with window refresh, this clipping function can be very handy for redraws of objects contained within windows. The error status of the routine is returned in the *status* variable. As with all of these routines, a status of 0 indicates that an error occurred, while a positive integer indicates that there was no error.

Another basic function is used to find the index number of the object located at a certain *x,y* coordinate on the screen. This is handy for determining whether the mouse pointer is positioned over an object when the user presses the button. This function is called *objc_find( ),* and its syntax is as follows:

```
int foundob, firstob, depth, x, y;
struct object tree[ ];
foundob = objc_find(tree, firstob, depth, x, y);
```

where *tree* is a pointer to the tree you want to search, *firstob* is the index number of the object you wish to begin the search with, and *depth* is number of generations down the tree you wish to search. The variables *x* and *y* hold the screen coordinates of the object you're searching for, usually the current *x* and *y* positions of the mouse pointer. If objc_find( ) locates an object at those coordinates, it returns the object number of the last object found in the foundob variable. If it doesn't find any objects there, it returns a value of −1.

The *objc_offset( )* function is the opposite of objc_find( ). It starts with the object number, and returns the absolute screen position of that object. This function is necessary because you can't determine the *x* and *y* coordinates of an object just by looking in the *ob_x* and *ob_y* fields of its object structure. The positions stored there are relative offsets from the object's parent, so to determine its absolute position, you must add all of the offsets of each preceding generation to the *x* and *y* position of the root. The syntax for the objc_offset( ) call is

```
int status, object, x, y;
struct object tree[ ];
status = objc_offset(tree, object, &x, &y);
```

where *tree* is a pointer to the object array, *object* is the index number of the object whose position is desired, and *x* and *y* are the variables in which that position is returned.

The *objc_change* function is used to change the *ob_status* field of an object, and, optionally, to redraw it at the same time. It's called like this:

```
int status, object, reserved, clipx, clipy, clipw, cliph, state, redraw;
struct object tree[ ];
status = objc_change(tree, object, reserved, clipx, clipy, clipw,
cliph, state, redraw);
```

where *tree* is a pointer to the object tree, *object* is the number of the object to be changed, and *state* is the new ob_state flag value. The *redraw* flag is used to indicate whether or not the object should be immediately redrawn; a 1 indicates that the object should be redrawn, while a 0 indicates that it should not. If the object is redrawn, the *clipx, clipy, clipw,* and *cliph* variables are used to set the clipping rectangle prior to the redraw.

The *objc_edit( )* function assists in the process of letting the user edit a text object. It's the low-level routine used by the dialog handler routine form_do( ) to implement text editing. The syntax of the objc_edit( ) call is

```
int status, object, char, index, type;
struct object tree[ ];
status = objc_edit(tree, object, char, &index, type);
```

where *tree* is a pointer to the object tree, and *object* is the number of the text object to be edited. The *char* variable contains the character to be inserted into the text string (usually obtained by reading the keyboard and using the result as the input here). The *index* variable does double duty. As input, it gives the routine the offset that tells it what place in the string is being edited. When the function concludes, it returns the new index number in this variable. Finally, the *type* variable contains a flag that indicates the type of editing function to be performed. The editing functions available are shown below:

Type Number	Macro Name	Description of Function
0	ED_START	Reserved for future use
1	ED_INIT	Combine the template string of TEDINFO field *te_ptmplt* with the text string of the *te_ptext* field to display the formatted string and then turn the cursor on
2	ED_CHAR	Check the input character against the validation string in TEDINFO field *te_pvalid,* update the *te_ptext* field if the input character is valid, and display the changed text
3	ED_END	Turn off the text cursor

You may notice that the C calling sequence for objc_edit( ) above differs from the Digital Research documentation. In that version, there are separate parameters for the text string index you input and the index returned by the program. The extra index pointer is added to the end of the parameter list. As of this writing, however, the *Alcyon* C bindings from Atari, and those derived from those bindings such as the *Megamax* compiler use the same index pointer for input and output. If you have problems making this function work correctly from C, you should examine either the source or a disassembly of your bindings, to determine the correct calling sequence.

In order to perform text editing, the Library function form_do( ) calls objc_edit with the function type set to ED_INIT to display the cursor in the first field to be edited. It then uses an *evnt_multi()* call to read the keyboard (among other things). When a key is pressed, it checks to see if it's one of the special keys, such as the cursor or Tab keys used to change to the next text field. If not, the string is edited by calling objc_edit( ) with the type flag set to ED_CHAR. If the field to be edited is changed, the cursor is turned off in the current field with the ED_END function of objc_edit, and the whole process is repeated with the next field to be edited.

The final three basic object functions allow you to reorder the linkages in the object tree array, without manually changing individual link fields. They are most useful for dynamically allocated object trees such as are used on the GEM

Desktop program, where it's not possible to determine ahead of time how many file icons will be needed in a window. If you set up your object trees ahead of time by using a resource file, as is usual, you won't have to worry about these calls.
The first operation is *objc_add()*, which is used to add an object as a child of a parent object.

```
int status, parent, child;
struct object tree[ ];
status = objc_add(tree, parent, child);
```

where *tree* is a pointer to the object tree, *parent* contains the object number of the parent object to which you wish to link the child, and *child* contains the object number of the child you wish to link in.

Likewise, you can remove an object from a tree altogether by using the *objc_delete()* function:

```
int status, object;
struct object tree[ ];
status = objc_delete(tree, object);
```

where *tree* is a pointer to the object tree and *object* contains the object number of the object to be removed.

Finally, you may change an object's order in relation to its siblings by using the *objc_order()* function:

```
int status, object, newpos;
struct object tree[ ];
status = objc_order(tree, object, newpos);
```

where *tree* is a pointer to the object tree, *object* is the number of the object to reorder, and *newpos* is a flag which indicates how to reorder this object's siblings. A value of 0 specifies that the object is to be moved to the end of the chain of siblings, 1 specifies that it be moved to a position second from the end of the chain, and so on. A value of −1 can be used to indicate that the object should be moved to the position of first sibling in the chain.

Program 4-1, written in C demonstrates the use of objc_draw( ) and objc_find( ), two of the more commonly used low-level object functions. It displays an object tree that consists of three objects—a root box, a text string, and a button. It then waits for a mouse button press and checks to see whether the mouse was over the button when the press occurred. If it was not, it waits for another button press. If the user clicks on the EXIT button, the program ends.

Program 4-1. object.c

```
/**********************************************************************/
/*                                                                    */
/*      OBJECT.C  -- Demonstrates the use of low-level               */
/*      Object Library functions on a pre-initialized                */
/*      object tree array.                                           */
/*                                                                    */
/*                                                                    */
/**********************************************************************/

#define APP_INFO " "
#define APP_NAME "Object Example"
#define WDW_CTRLS (NAME)
#define EXITBUTN 1

#include "aesshell.c"

char *strings[] = {
"EXIT",
"Push Button to Exit"};

OBJECT tree[] = {
{-1, 1, 2, G_BOX, NONE, OUTLINED, 0x21100L, 6,5, 22,10},
{2, -1, -1, G_BUTTON, SELECTABLE | DEFAULT | EXIT,
 NORMAL, strings[0], 12,6, 8,1},
{0, -1, -1, G_STRING, LASTOB, NORMAL,strings[1], 1,2, 21,1}
                };


demo()
{
  int msgbuf[8];
  int x, y, button, keys, object, state;

    evnt_mesage(&msgbuf);  /* skip the window redraw message, */

    for (x=0;x<3;x++) /* fix object x, y, width and height */
        {
        tree[x].ob_x *= cellw;
        tree[x].ob_y *= cellh;
        tree[x].ob_width *= cellw;
        tree[x].ob_height *= cellh;
        }
 /* draw the object tree */
    objc_draw(tree, 0, 1, work.g_x, work.g_y, work.g_w, work.g_h);

/*  check for mouse button press over EXIT button */
    do
    {
    evnt_button(1,1,1,&x, &y, &button, &keys);
    object = objc_find(tree, 0, 1, x, y);
    }
    while (object != EXITBUTN); /* keep checking 'til we find object 1 */

    /* show object selected */
    state = tree[EXITBUTN].ob_state ^ SELECTED; /* flip SELECTED bit */
    objc_change(tree, EXITBUTN, 0,
                work.g_x, work.g_y, work.g_w, work.g_h, state, 1);

}

/********************** End of Object.c **********************/
```

Please note two points about this program: First, an array of object structs called *tree* was declared and initialized at the same time. The *x*, *y*, width, and height values could not be filled, however, since there is no way to know in which of the three display resolution modes the program would run. To solve this problem, the number of character cells the object occupies was put into the *x*, *y*, width, and height fields. Then, when the program started, each of the *x*, *y*, width, and height values was multiplied by the default character cell width or height. This produced objects that occupied the same number of character cells regardless of the resolution mode (though these objects are twice as wide in low-resolution mode, since it only provides 40 characters per line).

Setting up objects in this manner may be easy to demonstrate, but it's by no means the preferred method. Typing all of the data by hand and then manually linking the objects to each other and to their ob_spec fields is nobody's idea of fun. As you'll see in the next chapter, there's a much easier and more effective way to create object trees.

For machine language programmers, Program 4-2 is a rough translation of the C sample program above. For the sake of brevity, the direct manipulation of the object *x*, *y*, width, and height fields was skipped in favor of using an AES Library routine, *rsrc_obfix()*, which does the same thing. This library routine will be discussed in the next chapter.

Program 4-2. object.s

```
*************************************************************
*                                                           *
*  OBJECT.S Demonstrates low-level object calls             *
*                                                           *
*************************************************************

*** External references

** Export:

.xdef     demo      * external demo subroutine.
.xdef     wdwctrl   * window controls
.xdef     wdwtitl   * window title string
.xdef     wdwinfo   * window info string

** Import:

.xref     aes     * the AES subroutine

.xref     ctrl0   * and the AES data arrays
.xref     ctrl1
.xref     ctrl2
.xref     ctrl3
.xref     ctrl4
.xref     addrin
```

```
.xref     aintin
.xref     aintout
.xref     workx   * and the window work rectangle
.xref     worky
.xref     workw
.xref     workh

  .text

demo:

*** Fix object position and size from characters to pixels

    move      #2,d4          * for three objects
    move      #114,ctrl0     * opcode = rsrc_objfix
    move      #1,ctrl1       * 1 intint
    move      #1,ctrl2       * 1 intout
    move      #1,ctrl3       * 1 addrin

    move.l    #tree,addrin * pass tree address
fixloop:
    move      d4,aintin      * use loop index for object number
    jsr       aes
    dbra      d4,fixloop     * decrement, and do again if not done

*** Draw the objects

    move      #42,ctrl0      * opcode = objc_draw
    move      #6,ctrl1       * 6 intin's

    move      #1,aintin+2      * draw down one level
    move      workx,aintin+4   * set clip x, y, w, h
    move      worky,aintin+6
    move      workw,aintin+8
    move      workh,aintin+10
    jsr       aes

*** Wait for button press

loop:
    move      #21,ctrl0        * opcode = evnt_button
    move      #3,ctrl1
    move      #5,ctrl2
    move      #0,ctrl3

    move      #1,aintin        * 1 click, left button only
    move      #1,aintin+2
    move      #1,aintin+4
    jsr       aes

*** See if button pressed while mouse was over EXIT object

    move      #43,ctrl0        * opcode = objc_find
    move      #4,ctrl1
    move      #1,ctrl2
    move      #1,ctrl3

    move      #0,aintin        * start searching at root
    move      aintout+2,aintin+4   * use mouse x and y for position
    move      aintout+4,aintin+6
    jsr       aes

    cmpi      #1,aintout       * object 1 found?
    bne       loop             * if not, check again
    rts


*** Storage space and data constants
```

```
.data
.even

wdwtitl:  .dc.b 'Object Example',0    * text of window title
wdwinfo:  .dc.b ' ',0                 * text of window info line
wdwctrl:  .dc.w 1                     * window control flag
exitstr:  .dc.b 'EXIT',0
pushstr:  .dc.b 'Push Button to Exit',0

tree:     .dc.l $FFFF0001, $00020014, $00000010, $00021100
          .dc.l $00060005, $0016000A
objc1:    .dc.l $0002FFFF, $FFFF001A, $00070000
          .dc.l exitstr
          .dc.l $000C0006, $00080001
objc2:    .dc.l $0000FFFF, $FFFF001C, $00200000
          .dc.l pushstr
          .dc.l $00010002, $00150001

.end
```

Chapter 5

Resource Files and Menus

Building an object tree by calculating the data necessary for each field of each structure in the array can be a long and tedious process. Each object requires 11 pieces of data. One of those is an *ob_spec* field, which may be a pointer to another data structure such as an ICONBLK which requires another 14 items of information, two of which are pointers to image data arrays. This means that just one object can have a large quantity of data associated with it. And GEM programs don't just use one or two objects; they use dozens. Alerts, dialog boxes, menus—all are built of many, many objects.

There are other problems with building objects out of data as well. For one thing, it's difficult to picture the size and relationship of objects based on mere numbers. If the object doesn't look quite right, you've got to change the numbers and recompile to try a new combination. Moreover, as you've seen from the example program at the end of the previous chapter, you have to change the size and position fields for each object based on the current display resolution mode. Finally, you have to calculate all the links that define their various parent-child relationships.

Because objects are an integral part of GEM, and because they're so difficult to build from data, Digital Research developed a system which can create object data. The system makes use of Digital Research's *Resource Construction Set* or a similar program. The *Resource Construction Set* is one of the best features of GEM. It takes the effort out of creating objects, and it allows you to position them relative to one another, change their parent-child relationships, enter text, and set the various attribute flags and object state flags for each object.

Once you've created the object data with the program, you may save it to a resource file. This is a data file ending with the .RSC extender. For instance, STBASIC.RSC is the resource file for the STBASIC.PRG program. When the resource file is created, the AES Resource Library routines enable your program to load the file, create the object structure data array from the information loaded, and find the addresses of individual object trees within that array.

Using resource files and a resource construction program has several fringe benefits besides making the object creation process quick and easy. First, it helps make the program more portable, both to other machines (such as the IBM PC) and within the various display resolution modes of the ST itself. As you'll see later, size and position information are stored within resources as character positions and are translated to display-specific figures automatically when the resource is loaded, much as was done manually in the example program in the last chapter.

Another advantage is that the programmer can experiment with the size, placement, and characteristics of objects without having to recompile each time. All you have to do is edit the resource file and run the program to see how the changes work.

Finally, the use of resource files makes it easy to create foreign-language versions of your program without recompiling. In fact, since the user can edit the resource file, the language in menus, dialog boxes, and alerts may be changed without any need for programming experience. This makes it possible for the program to be customized by the user. For these reasons, it's generally preferable to use resource files, even if for some reason you think it would be more convenient to create the object trees within the source code of the program itself.

Resource Construction Programs

Not every C compiler or assembler for the ST comes with a resource construction program, but no GEM programmer should be without one.

The original *Resource Construction Set (RCS)* is included with the ST Developers Kit available from Atari, along with the *Alcyon* C compiler. The *Megamax* C compiler package includes the *Megamax Resource Construction Program (MMRCP).*

If you aren't using either of those development systems, Kuma Computers Ltd. market a separate resource construction program called *K-Resource (KRSC),* which can be used with any compiler or assembler. Each of these programs has slightly different features. For example, *KRSC* and *MMRCP* both have built-in icon image editors, while the *RCS* does not. However, *RCS* and *MMRCP* both allow the user to load image data that was created using other programs such as *Icon Editor, Professional Icon Editor,* or *Degas Elite,* while *KRSC* does not. Under most circumstances, however, the differences between these programs is not significant. All of them perform very well at the basic task of constructing a resource file.

To build a resource with one of these programs, open a resource window, and drag one of several different tree icons onto the window. These icons represent various types of object trees, such as menus or dialogs.

The distinctions among types of object trees are strictly for the purposes of the resource construction program itself; no actual difference exists in the object structure of these trees. But such distinctions can help make the resource programs more effective. For example, when you select a menu tree, the program presents a prepared template of text objects arranged as in a typical menu, and the program restricts your selection to the type of text objects found within menus. The dialog tree can contain any object, but all objects are positioned so that their borders are aligned with even text character positions. The free or panel tree can contain any object at any pixel position.

The various resource programs also allow you to edit alert strings, as well as free text strings and free images (those not contained within an object tree).

When you drag a tree icon to the resource window, you create the root object for that tree (usually a G_BOX with an outlined border). To add children to the tree, you drag icons representing the various object types to a position in which they're completely enclosed by the root object.

You may create children for these objects also, by dragging additional icons into their rectangles. Once you've added these objects to the tree, edit them as you see fit. You can drag them to a new position with the mouse, change their size by dragging their lower right hand corner, or edit their attribute and flags. Simply double-click on the object to open a dialog box. From that dialog box you can set or reset each flag individually. (In *RCS* version 2, you highlight the icon by clicking on it and then select the appropriate flags from pop-up menus located on a control panel.) This allows you to experiment with the size, placement, and attributes of the objects interactively, immediately viewing any changes you make.

When you're finished with one object tree, you can add more trees until you create all the objects used in your program. When you're satisfied with the results, you may save the resource file. In addition, if you've created names for the various objects, the program will save those in a separate file (for future editing sessions), and it will also save a header file containing macro definitions which match the names to the corresponding object numbers. *The Resource Construction Set* also allows you to save a file containing C language definitions for all of the objects and their associated data structures.

Structure of Resource Files

Many of the fields included in object structures contain the absolute memory addresses of other data structures. But it's impossible to know in advance the absolute memory address at which the resource file will be loaded, because that depends on the program size and which desk accessory programs and fonts are loaded. Therefore, in addition to the object structure data, and associated data structures, the resource file must also contain information that allows the AES to patch in the absolute addresses where necessary. This information is contained in the resource header, a data structure located at the very beginning of each resource file. This structure consists of an array of words containing the size of the other data structures, and their offsets within the resource file. Its C language definition is

```
int rsh_vrsn        /* version number */
int rsh_object      /* object block offset */
int rsh_tedinfo     /* TEDINFO block offset */
int rsh_iconblk     /* ICONBLK block offset */
int rsh_bitblk      /* BITBLK block offset */
int rsh_frstr       /* Free String block offset */
int rsh_string      /* String block offset */
int rsh_imdata      /* image data block offset */
int rsh_frimg       /* free image data block offset */
int rsh_trindex     /* tree index block offset */
int rsh_nobs        /* number of objects */
int rsh_ntree       /* number of trees */
int rsh_nted        /* number of TEDINFOs */
int rsh_nib         /* number of ICONBLKs */
int rsh_nbb         /* number of BITBLKs */
int rsh_nstring     /* number of strings */
int rsh_nimages     /* number of images */
int rsh_rssize      /* total size of the resource, in bytes */
```

Following the resource header are the actual data arrays that make up the objects. First comes an array of strings, followed by an array of BITBLKs, an array of image data, an array of ICONBLKs, an array of TEDINFOs, and finally the array of OBJECTs.

There are important differences between the data arrays stored in the resource file and those stored in the actual object array used by the AES Object Library routines.

First, as noted above, the resource file doesn't contain the actual pointers used by the ob_spec field or other data structures. Instead, these fields contain an index number which indicates the position of the desired data structure within its array. For example, the ob_spec field of a G_STRING contains the index number of the desired string within the string data array instead of the address of that string.

Another difference is that screen *x*, *y*, width, and height values are stored in character units rather than pixel units. For example, if a G_BOX starts at position (24,32) and each character is eight pixels high and eight pixels wide, the position stored in the resource is (3,4). This allows the AES to draw the object to scale, regardless of the screen display mode in effect at load time (assuming an 80 × 25 character display). If the object wasn't created on a character boundary, the low byte of each rectangle word contains the closest character position, while the high byte contains the number of pixels left over.

The last data structure in the resource file is another one used by the AES when loading the font. This structure is called the Tree Index, and it contains the index number of each tree's root object in the object array. This allows the AES to find the starting address of each object tree in the object array.

Loading a Resource

In order to use the object trees stored in a resource file, your program must first load the resource. The AES provides a special Library routine for just this purpose, called *rsrc_load( )*. The calling sequence for this function is

```
int status
char *filename;
status = rsrc_load(filename);
```

where *filename* is a pointer to a null-terminated string that contains the name of the resource file. This name is usually the same as that of the application, only with an extender of .RSC instead of .PRG. For example, RCS.RSC is the resource file for the *Resource Construction Set* program (filename RCS.PRG).

If you don't specify a directory path, GEMDOS will look for the the resource file in the same directory as the application, which is the usual state of affairs. If the rsrc_load( ) routine is unable to find the resource file or is unable to load it properly for any other reason, a value of 0 is returned in the status variable. Your program will want to check this variable after attempting a resource load, since you'll want to notify the user with an alert box and terminate the program at once if the resource file didn't load properly.

When the AES loads a resource file, it finds the size of the file, allocates enough free memory to hold it, and then reads the contents of the file into the allocated space.

Next, the AES changes the data structures that were loaded into memory. It converts the size and position values stored in character units back into pixel units, based on the current default character size. It replaces the array offset values in the OBJECT, TEDINFO, INCONBLK, and BITBLK structures with actual address pointers. It replaces the array offset values in the Tree Index with actual address pointers to the beginning of each object tree. Then, it stores the address of the Tree Index array in the *ap_ptree* field of the application's Global data array.

Since the rsrc_load( ) process allocates some of the computer's free memory, you should always remember to give that memory back when you end your program. And though your program will only load one resource file in most cases, if you want to load more than one, you must unload the first before loading the second. In either case, the call you use is *rsrc_free( )*:

```
int status;
status = rsrc_free( );
```

Once you've loaded a resource file, you can find out the addresses of the various data structures it contains by using the *rsrc_gaddr( )* Library call. The syntax for this call is

```
int status, type, index;
long address;
status = rsrc_gaddr(type, index, &address);
```

where *type* specifies the type of data structure, *index* gives the position within the data array, and *address* is the variable which holds the address to be placed in the data structure. The types of data structures whose addresses you may find with this function, and the macro names assigned them in the GEMDEFS.H header file, are as follows:

Type Number	Macro Name	Data Structure
0	R_TREE	Object tree
1	R_OBJECT	OBJECT
2	R_TEDINFO	TEDINFO
3	R_ICONBLK	ICONBLK
4	R_BITBLK	BITBLK
5	R_STRING	Pointer to free strings
6	R_IMAGEDATA	Pointer to free image data
7	R_OBSPEC	Ob_spec field of OBJECT
8	R_TEPTEXT	Te_ptext field of TEDINFO
9	R_TEPTMPLT	Te_ptmplt field of TEDINFO
10	R_TEPVALID	Te_pvalid field of TEDINFO
11	R_IBPMASK	Ib_pmask field of ICONBLK
12	R_IBPDATA	Ib_pdata field of ICONBLK
13	R_IBPTEXT	Ib_ptext field of ICONBLK
14	R_BIPDATA	Bi_pdata field of BITBLK
15	R_FRSTR	Ad_frstr—the address of a pointer to a free string
16	R_FRIMG	Ad_frimg—the address of a pointer to a free image

In practice, you'll be using rsrc_gaddr( ) most often for data structure type R_TREE, the object tree. Once you have the address of the object tree, you can use the object index numbers to access the individual objects.

Another type that you may use is R_STRING, which can be used to get a pointer to an alert string. Alert strings are stored as free strings by the resource construction programs. The other data structure types are there primarily for the AES, which uses them when it fixes pointers in those data structures at resource load time. When you're using rsrc_gaddr( ), remember that the input value *&address* is a pointer to a long value (which itself may be a pointer). So if you substitute a pointer for the long value address, you still must use a pointer to that pointer for the input value as shown below:

```
OBJECT *tree;
rsrc_gaddr(R_TREE, MENUTREE, &tree);
```

Two Resource Library functions remain to be discussed. Though these functions are used primarily by the AES during the resource load process, you may find use for them in your applications. The first is *rsrc_objfix( ),* and it's used to convert an object's position and size values from character units to pixels. The syntax for this function call is

```
int status, object;
struct object tree[ ];
status = rsrc_obfix(tree, object);
```

where *tree* is a pointer to the array of object structures, and *object* is the index number of the object whose position and size is to be converted. While this function is mostly used by the AES when it loads a resource file, it could have been used in the example in the previous chapter where an object was created out of initialized data structures. Here is how the program could have been modified to take advantage of this function. This is the section of the program where the position and size of the objects were fixed:

```
for (x=0;x<3;x++) /* fix object x, y, width, and height */
   {
   tree[x].ob_x *= cellw;
   tree[x].ob_y *= cellh;
   tree[x].ob_width *= cellw;
   tree[x].ob_height *= cellh;
   }
```

to this:

```
for (x=0;x<3;x++)   /* fix object x, y, width, and height */
   rsrc_objfix(tree, x);
```

In fact, that's just what was done in the machine language version of the example.

The final call is used to store the address of a data structure array element in memory. It's called *rsrc_saddr,* and its calling sequence is

```
int status, type, index;
long address;
status = rsrc_saddr(type, index, address);
```

where *type* specifies the type of data structure, *index* gives the position within the data array, and *address* is the variable which holds the address to be placed in the data structure.

This function is used by the AES when it corrects the address pointers at resource load time, but there are instances when you may want to use it yourself in a program.

Though resource files are made to be as independent of the display resolution as possible, there are some situations where objects may not work equally well in all resolutions.

The first is when using the low-resolution display mode. The resource construction programs are designed to run in medium- or high-resolution modes, so most resource files are built on the assumption that an 80 × 25 character display will be used. If this isn't the case, parts of large object trees might be lost off the edge of the screen.

The second area of possible incompatibility concerns the aspect ratio of the display. Though objects are lined up by character positions, the actual image data of G_IMAGE or G_ICON object consists of pixels. If the aspect ratio of the display is different from the one on which it was created, the icons and images may not look right and may not be located in the right spot. For example, the Desktop program icons for disks, files, and folders look fine in high- or low-resolution modes, but in medium-resolution they appear to be tall and skinny.

There are several ways to deal with this problem. First, you can try to make your resource display-independent when you create it, by keeping object widths smaller than forty characters, and using compromise images that are a little shorter and fatter in high resolution than you'd normally make them, and a little taller and skinnier in medium resolution.

The second approach is to create separate resource files for the different resolutions. To some extent, this goes against the basic philosophy of compatibility behind the resource files.

A third alternative is to place extra image and icon data in the resource and patch the appropriate data into the data structures whenever the program comes up in a display resolution mode for which the object trees weren't designed. In this last case you might find the rsrc_gaddr( ) and rsrc_saddr( ) calls handy. You can use rsrc_gaddr( ) to find the addresses of the alternate data structures and rsrc_saddr( ) to patch them into the existing ones.

Menus

One of GEM's most helpful features is its system of drop-down menus. These menus are composed of objects whose tree is usually loaded as part of the resource file. Once the application installs the menu, the menu titles—which display the broad categories of selections available—appear in the menu bar at the top of the screen.

When the user moves the mouse pointer over one of these titles, the Screen Manager saves the old screen background in the menu buffer and draws a box beneath the title. This box contains one or more menu items, which represent program options that the user may select.

As the user moves the mouse pointer over a selectable item, its text is drawn in reverse video to highlight it. The user can click the mouse button over one of these highlighted items to pick it. When this happens, the Screen Manager restores the background display from the menu buffer and sends a message to the application telling it which menu title and menu item were selected. The title belonging to the item you selected is left highlighted while your program handles the message. If the user decides not to select a menu item, he or she may get rid of the menu either by replacing it with another or by moving the mouse pointer off the menu bar entirely and clicking the left mouse button.

All of the data for a menu is contained in a normal GEM object tree. But since the AES Screen Manager does so much autonomous manipulation of menus, this object tree must follow a set format.

Since the menu system uses objects that can be located in the menu bar or anywhere else on screen, the root object of the menu is a G_BOX that covers the entire display area of the screen. This root object has exactly two children, whose rectangles cover the root object completely. The first, called the BAR, is a G_BOX that covers the whole menu bar at the top of the screen. The second, called the SCREEN, is a G_IBOX that covers the rest of the display area, excluding the menu bar.

The BAR and the ACTIVE. The BAR is 80 characters wide and is one character plus two pixels high. It contains exactly one child, a G_IBOX called the ACTIVE. The ACTIVE covers only the portion of the menu bar that actually contains menu title objects. It takes its name from the fact that the

Screen Manager activates a menu whenever the mouse pointer enters its rectangle. The ACTIVE has as its offspring the various G_TITLE objects that represent the menu titles. These objects should line up side to side and completely cover the ACTIVE.

The SCREEN. The other child of the root object, the SCREEN, is the parent of the box objects which contain the drop-down menus that appear under the menu titles. Each drop-down is a separate G_BOX child of the SCREEN. No single drop-down box can be larger than one quarter of the screen display in size, since that's the size of the offscreen buffer where the AES temporarily stores what's in the screen display behind the menu. The drop-down boxes each contain a number of objects, usually G_STRINGS, that represent the individual menu items. These objects should completely cover the drop-down's rectangle.

Because of the rigid hierarchy requirements of menu object trees, it would be extremely difficult to construct one without using a resource construction program.

When you open a menu-type tree with a resource construction program, it automatically creates the root object, the BAR, the SCREEN, the ACTIVE, and a couple of default menu titles containing appropriate menu items. Whenever you create a new menu title the program changes the size of the ACTIVE, moves the new title so that it's adjacent to previous ones, and creates the drop-down box with space for a single menu item.

In return for this convenience, however, these programs enforce some restrictions. Most only allow you to use G_STRING-type objects for menu items. While G_STRINGs are the preferred choice for this task, because it's easy for the AES to draw them quickly, there are some cases where you might prefer to use box-type objects, or even icons.

In order to add non-G_STRING objects as menu items with the resource construction programs, you must first create the menu tree, then close it, and change its tree type from menu to dialog. At that point, you can enlarge the drop-down boxes and add any object that you wish. Each object, however, should still stretch across the entire width of the drop-down box. If you want to place two or more siblings side by side on a line, make sure that you cover them with another sibling that occupies the whole line, so that they all get selected at the same time when that menu item is chosen. If any of the drop-down box is left exposed, the whole box will be inverted when the user moves the mouse over that part.

In a dialog tree, you'll find that all of the objects are drawn at once, making it difficult to get at some of the overlapping siblings. By setting the HIDETREE attribute flag, you should be able to temporarily remove those that are in your way and then reveal them when you're done.

Once you've finished editing the tree, you should be able to close it and to change it back to a menu tree. If you've obeyed the rules, the resource construction program should let you make the change.

There are certain menu conventions which GEM programs should follow, in order to promote uniformity. The first menu title is customarily set to DESK. The DESK drop-down box must contain exactly eight menu items. The first is usually titled *About Program . . .*, where Program is filled in with the name of the application. Note the three dots at the end of this item. This is another GEM convention, which tells the user that selecting this item leads to a dialog (in this case, the dialog box which displays the program credits). The next item is a line of faint dashes, created by setting the DISABLED flag. This line is a conventional device called a separator bar, which sets off one group of menu options from another. The other six objects are dummy strings which the AES fills in with the names of desk accessories that have registered their menu entries.

The next menu title should be FILE. The drop-down for this title should contain entries like New, Open . . . , Save, Save As . . . , and Quit. Some of these items may contain a symbol next to the text, such as *Open . . . ^O*. This shows the user that there are keyboard equivalents to this menu item. In the example shown, the user could hold down the Control key and press the letter *O*, and it would have the same effect as if the Open . . . item had been selected from the menu.

The next title to use, if appropriate, is EDIT. This contains items such as Cut, Copy, Paste, and Delete. From there on, the menu title selection is up to you. One thing that you should keep in mind when creating menus, however, is to keep them short. If you find that you need more than eight or nine menu items, you may want to consider going to a dialog box instead of a menu.

Using Menus

Once you've created your menu tree with the resource construction program of your choice, the next step is installing it from your program. After you've loaded the resource file with *rsrc_load( ),* the next step is to retrieve the address of the menu tree with the *rsrc_gaddr( )* call:

```
long menuaddr;
rsrc_gaddr(R_TREE, MENUTREE, &menuaddr);
```

R_TREE is a macro name defined in GEMDEFS.H, and MENUTREE is a macro name for the root object of your menu tree, defined in the .H file created by the resource construction program. When you've got the address of the menu tree, you're ready to tell the AES to install your menu. You do this using the *menu_bar( )* call, whose syntax is as follows:

```
int status, showflag;
OBJECT *tree;
status = menu_bar(tree, showflag);
```

where *tree* is a pointer to an object tree (in the example above, you would use *&menuaddr* for that pointer). *Showflag* indicates whether you want to draw the menu bar or erase it. When you wish to install the menu bar, set showflag to 1. At the end of your program, before calling *appl_exit,* you should erase the menu bar by calling menu_bar( ) once again, only this time, with showflag set to 0.

Once you've called menu_bar( ), the AES draws your menu bar at the top of the screen and begins to handle user interaction with the menus. When the user moves the mouse pointer to the ACTIVE, the AES drops down the proper menu. When the user selects a menu item, the AES sends your program message number 10, MN_SELECTED. The format for this message is

Word Number	Contents
0	10 (MN_SELECTED), a menu item was selected by the user
3	The object number of the menu title that was selected
4	The object number of the menu item that was selected

When your program receives this message (via an *evnt_multi* or *evnt_message* call), you'll generally check word 4 of the message buffer to see what item was selected and take the appropriate action. While you are performing whatever task the user has selected, the AES leaves the menu title display highlighted to show that the program is busy processing the menu request. When you've finished, you must change the title back to its normal video display with the *menu_tnormal* function. The calling format for this function is

int status, title, setting;
OBJECT *tree;
status = menu_tnormal(tree, title, setting);

where *tree* is a pointer to the menu object tree, *title* is the object number of the title, and *setting* is a flag indicating how you want that title displayed. If setting contains a value of 1, the title will be drawn in normal video. If setting is 0, the title will be highlighted in reverse video. Under normal circumstances you'll be using word 3 of the message buffer for the title object number and a setting of 1.

Program 5-1 demonstrates some simple menu handling. It sets up a menu with three titles and four selectable items. The DESK menu contains one item: About Menu1. . . . The FILE menu contains one item: Quit ^C. The DRAW menu contains two items: Pattern 1 [F1] and Pattern 2 [F2]. The About item on the DESK menu is used to display an alert box (those will be covered in more detail in the next chapter), the two DRAW items draw patterned ellipses in the program window, and the Quit item is used to exit the program. Because this is a simplified example, the *handle_msg()* routine only deals with menu messages. In an actual application, your program would have to deal with all of the window messages as well.

Program 5-1. menu1.c Program

```
/*************************************************************************/
/*                                                                       */
/*      MENU1.C -- Demonstrates handling of simple menus,                */
/*      with keyboard equivalents.                                       */
/*                                                                       */
/*************************************************************************/

#define APP_INFO " "
#define APP_NAME "Menu Example 1"
#define WDW_CTRLS (NAME)

#define MOUSE_OFF graf_mouse(256,&dummy)
#define MOUSE_ON  graf_mouse(257,&dummy)
#define CTRLQ 0x1011  /* keycode for CTRL-Q key combo */
#define F1KEY 0x3B00  /* keycode for function key 1 */
#define F2KEY 0x3C00  /* keycode for function key 2 */

#include "aesshell.c"
#include "menu1.h"   /* include file from RCS */
```

```
int dummy;
OBJECT *menutree;
char *alert;

demo()
{

int event,done = 0;
int key, msg[8];

if (!rsrc_load("MENU1.RSC"))      /* Load resource file */
   {
   form_alert(3,"[0][Fatal Error!Can't find MENU1.RSC file!][Abort]");
   return(0);                     /* Abort if it's not there */
   }

/* get address of menu tree and alert string */
rsrc_gaddr(R_TREE, MENUTREE, &menutree);
rsrc_gaddr(R_STRING, ABTALERT, &alert);

MOUSE_OFF;                     /* Hide the mouse pointer */
menu_bar(menutree,1);          /* Show the menu bar */
MOUSE_ON;                      /* Show the mouse pointer */

vsf_interior(handle,2);  /* Set Fill Pattern for Ellipse */

/* Main Program Loop */

   while (! done)   /* until user selects "Quit" item */
   {                /* check menus and keyboard */
   event = evnt_multi(MU_MESAG|MU_KEYBD,
   0,0,0,           /* evnt_button */
   0,0,0,0,0,       /* evnt_mouse1 */
   0,0,0,0,0,       /* evnt_mouse2 */
   &msg,            /* evnt_mesg */
   0,0,             /* evnt_timer */
   &dummy,&dummy,   /* mouse x,y */
   &dummy,          /* mouse button */
   &dummy,          /* shift keys */
   &key,            /* evnt_keyboard */
   &dummy);         /* number of clicks */

   if (event & MU_MESAG)  /* if we get a message, handle it */
      done = handle_msg(msg);

   if (event & MU_KEYBD) /* if key, check for equivalents */
      done = handle_key(key);

   } /* end of main WHILE loop */

menu_bar(menutree,0);          /* Remove the menu bar */

} /* end of DEMO function */

/* Message Handler routine -- only handles menu messages */
/* (yours should also handle redraws, window topping, etc. */

handle_msg(msg)
int msg[8];
{

   int done=0;

   switch (msg[0])        /* check message type */
   {
   case MN_SELECTED:      /* if menu message type */
      switch (msg[4])     /* check menu item */
      {
```

```
      case ABOTITEM:   /* if About... display alert */
         form_alert(0,alert);
         break;
      case PAT1ITEM:   /* draw Pattern 1 */
         draw(7,2);
         break;
      case PAT2ITEM:   /* draw Pattern 2 */
         draw(5,3);
         break;
      case QUITITEM:   /* Quit */
         done = 1;
         break;
      default:
         break;
      }  /* end of switch on menu item */
      menu_tnormal(menutree, msg[3], 1); /* set menu to normal */
      break;

   default:
      break;
   }/* end of switch on message type */

   return(done);          /* report done status */
}

/* Keyboard Handler routine -- checks for keyboard */
/* equivalents of menu selections */

handle_key(key)
int key;
{

   int done = 0;

   switch (key)      /* check key code */
   {
   case F1KEY:       /* draw Pattern 1 */
      draw(7,2);
      break;
   case F2KEY:       /* draw Pattern 2 */
      draw(5,3);
      break;
   case CTRLQ:       /* Quit */
      done = 1;
      break;
   default:
      break;
   }  /* end of switch on key */

   return(done);
}

/* Ellipse Drawing Routine */

draw(pattern, color)
int pattern, color;
{
      vsf_style(handle,pattern);
      vsf_color(handle,color);
      MOUSE_OFF;               /* Hide the mouse pointer */
      v_ellipse(handle, work.g_x+work.g_w/2, work.g_y+work.g_h/2,
               work.g_w/2, work.g_h/2);
      MOUSE_ON;                /* Show the mouse pointer */

}

/* ****************** End of Menu1.c ************** */
```

In order to run this program, you must first create a resource file called MENU1.RSC. If you have a resource construction program, you'll need to create two object trees. The first is a menu tree with the C macro name MENUTREE. This menu has three titles. To the default DESK and FILE titles add a DRAW title. Under the DESK menu, edit the first item's string to read About Menu1 . . . and give it the C macro name ABOTITEM. Edit the item under the FILE menu to read Quit ^Q and give it the name QUITITEM. Next, add two items under the DRAW menu. The first reads Pattern 1 [F1] and has the name PAT1ITEM. The second reads Pattern 2 [F2] and has the name PAT2ITEM. The second object tree is an alert string called ABTALERT. Drag four strings to the alert window. The first line reads *Menu demo with multi-object,* the second reads *items and keyboard equivalents,* and the third reads *Select "Quit" to end.* The fourth line is made up of dashes. The string in the button should be edited to read *I'll remember that.*

If you don't have a resource construction program, you should get one without further delay, but in the meantime, you'll be able to build the resource file needed for this example by running the RSCBUILD program in Appendix C.

The resource file that we created for this program was a bit fancier than the one described above. The menu items Pattern 1 and Pattern 2 each include a colored G_BOX which displays the pattern fill. To create this kind of a menu item, you must first change the type of the tree from menu to dialog in your resource construction program.

When you display this tree, all the drop-downs will be visible. Get a hold of the drop-down box for DRAW by holding down the Control key and holding down the mouse button in the lower right corner of the box. This selects the parent object. Drag the box to the right so that you make some room for a G_BOX. At this point, it's probably a good idea to move the second menu item to the clipboard, temporarily. Next, add a G_BOX that completely covers the space to the right of the G_STRING on the top line. Make it foreground color 2, fill style 6, with an outside border of one pixel. Next, move a G_IBOX to the second line and size it so that it covers the entire menu line. Edit it to remove the border and move it over the top menu line. If your editor tells you that the G_IBOX now covers its two siblings, and asks if you want to make it their parent, answer "No." You want the three objects to remain siblings, so that when the user selects the IBOX, all will be highlighted. Repeat the process with the second menu item (its G_BOX has a foreground color of 2 and a fill pattern of 5), and you've got a menu with words and pictures on the same line.

Once you've set up your menu, there are a number of ways you may modify it. One way you can change a menu once your program is running is to disable or reenable one or more menu items. Disabling a menu item signals the user that a choice is temporarily inappropriate. For example, if the user starts a new project which hasn't yet been named, you might not want to allow the Save option to be used, forcing the user to use Save As . . . . And if the user hasn't entered any information since starting the project, you might want to prevent the use of either Save or Save As . . . . To disable a menu, you use the *menu_ienable( )* function, whose syntax is as follows:

```
int status, item, setting;
OBJECT *tree;
status = menu_ienable(tree, item, setting);
```

where *tree* is a pointer to the menu tree, *item* is the object number of the menu item to change, and *setting* is a flag which indicates whether you wish to enable or disable that menu item. A setting of 0 indicates that you want the menu item disabled, while a setting of 1 directs the AES to reenable that item. What the menu_ienable( ) call actually does is change the DISABLED flag. If you prefer, you may also change this by using *objc_change( ),* or by writing a new value directly to the *ob_state* member of the object structure. When the DISABLED bit is set, the AES draws the affected menu item faintly and won't allow the user to select that item.

The menu_ienable call normally works for menu items only and doesn't work for menu titles. There is, however, an undocumented feature in the current version of GEM which allows you to disable an entire title. If you call menu_ienable with the top bit of object number set to 1, the menu title will be drawn faintly, and the AES won't drop down its menu box when the user moves the mouse pointer over it. Since this is an undocumented feature which may not work reliably, use it with caution and don't be surprised if it's changed in future versions of the operating system.

Some menu selections represent Boolean "on-off" types of choices. For example, a text-editing program might allow you turn a word-wrapping feature either on or off. If your menu item allows the user to toggle this feature from on to off to on again, there's got to be some way of knowing what the current state of affairs in order to determine whether or not to toggle the item. One way of letting the user know is to put a checkmark next to the text of the menu item when it's selected and to erase it when it's not selected. The *menu_icheck* function allows you to either place a checkmark next to the text of a menu item or to erase the checkmark. The syntax for this call is

```
int status, item, setting;
OBJECT *tree;
status = menu_icheck(tree, item, setting);
```

where *tree* is a pointer to the menu object tree, *item* is the object number of the menu item, and *setting* is a flag indicating whether you want to place the checkmark or remove it. A setting of 1 adds the checkmark, and a setting of 0 removes it. Since the checkmark is drawn at the left side of the text box, you should always leave one or two blank spaces in front of the item's text (this will make the menu look better whether or not you use checkmarks). As with menu_ienable( ), menu_icheck( ) actually changes the setting of a single bit in the *ob_state()* word. In this case, it's the CHECKED flag, and, if you prefer, you may change this flag directly as well.

Another way to indicate the current setting is to change the text of the menu item itself. Typically, if the option is on, the menu item will read *Turn this option off*, and when it's off, it will read *Turn this option on*. You can change the text of a G_STRING menu item by using the *menu_text* call:

```
int status, item;
char *text;
OBJECT *tree;
status = menu_text(tree, item, text);
```

where *tree* is a pointer to the menu object tree, *item* is the object number of the text object to change, and *text* is a pointer to the replacement string. The size of the replacement string should be the same as the original one. If it's longer, it may go out of the menu box and onto the desktop. If you're planning to use replacement strings, include enough space at the end of each menu item so that the drop-down box will be large enough to accommodate the longest string.

Program 5-2 demonstrates how to handle checkmarks, disabled menu items, and menu items with alternating text.

Program 5-2. menu2.c

```
/*************************************************************************/
/*                                                                       */
/*      MENU2.C -- Demonstrates handling of check marks,                 */
/*      disabled menu items, and alternate text.                         */
/*                                                                       */
/*************************************************************************/

#define APP_INFO " "
#define APP_NAME "Menu Example 2"
#define WDW_CTRLS (NAME)

#define MOUSE_OFF graf_mouse(256,&dummy)
#define MOUSE_ON  graf_mouse(257,&dummy)
#define CTRLQ 0x1011  /* keycode for CTRL-Q key combo */

#include "aesshell.c"
#include "menu2.h"   /* include file from RCS */

int dummy, key, msg[8];
char *alert, *menuon, *menuoff;
OBJECT *menutree;


demo()
{

int event, done = 0;

if (!rsrc_load("MENU2.RSC"))       /* Load resource file */
   {
   form_alert(3,"[0][Fatal Error|Can't find MENU2.RSC file!][Abort]");
   return(0);                      /* Abort if it's not there */
   }

/* get address of menu tree,alert and item strings */
rsrc_gaddr(R_TREE, MENUTREE, &menutree);
rsrc_gaddr(R_STRING, ABTALERT, &alert);
rsrc_gaddr(R_STRING, ONSTRNG, &menuon);
rsrc_gaddr(R_STRING, OFFSTRNG, &menuoff);

MOUSE_OFF;                      /* Hide the mouse pointer */
menu_bar(menutree,1);           /* Show the menu bar */
MOUSE_ON;                       /* Show the mouse pointer */

/* Main Program Loop */

   while (! done)   /* until user selects "Quit" item */
   {                /* check menus and keyboard */
   event = evnt_multi(MU_MESAG|MU_KEYBD,
   0,0,0,           /* evnt_button */
   0,0,0,0,0,       /* evnt_mouse1 */
   0,0,0,0,0,       /* evnt_mouse2 */
   &msg,            /* evnt_mesg */
   0,0,             /* evnt_timer */
   &dummy,&dummy,   /* mouse x,y */
   &dummy,          /* mouse button */
   &dummy,          /* shift keys */
   &key,            /* evnt_keyboard */
   &dummy);         /* number of clicks */
```

```
    if (event & MU_MESAG)   /* if we get a message, handle it */
       done = handle_msg();

    if (event & MU_KEYBD) /* if key, check for equivalents */
       if(key == CTRLQ)
          done = 1;

    } /* end of main WHILE loop */

menu_bar(menutree,0);       /* Remove the menu bar */

} /* end of DEMO function */

/* Message Handler routine -- only handles menu messages */
/* (yours should also handle redraws, window topping, etc. */

handle_msg()
{

    int done=0;

    switch (msg[0])       /* check message type */
    {
    case MN_SELECTED:     /* if menu message type */
       switch (msg[4])    /* check menu item */
       {
       case ABOTITEM:     /* if About... display alert */
          form_alert(0, alert);
          break;
       case CHEKITEM:     /* Toggle Checkmark */
          if(menutree[CHEKITEM].ob_state & CHECKED)
             {
             menu_icheck(menutree, CHEKITEM, 0);
             text("Check Mark is now turned OFF",1);
             }
          else
             {
             menu_icheck(menutree, CHEKITEM, 1);
             text("Check Mark is now turned ON",2);
             }
          break;

       case TOGLITEM:     /* draw Pattern 2 */
          if(menutree[ABLEITEM].ob_state & DISABLED)
             {
             menu_ienable(menutree, ABLEITEM, 1);
             menu_text(menutree, TOGLITEM, menuoff);
             text("Menu Item is now turned ON",2);
             }
          else
             {
             menu_ienable(menutree, ABLEITEM, 0);
             menu_text(menutree, TOGLITEM, menuon);
             text("Menu Item is now turned OFF",1);
             }
          break;

       case ABLEITEM:     /* draw Pattern 2 */
          text("Thanks for turning me on",1);
          break;

       case QUITITEM:     /* Quit */
          done = 1;
          break;
       default:
          break;
       }  /* end of switch on menu item */
       menu_tnormal(menutree, msg[3], 1); /* set menu to normal */
       break;
```

```
   default:
      break;
   }/* end of switch on message type */

   return(done);         /* report done status */
}


/* Print Text Routine */

text(string, color)
int  color;
char *string;
{
      vst_color(handle,color);
      MOUSE_OFF;                     /* Hide the mouse pointer */
      clear_rect(&work);              /* clear the area */
      v_gtext(handle, 10*cellw, 20*cellh, string);
      MOUSE_ON;                      /* Show the mouse pointer */

}

/* ************************* End of Menu2.c ******************** */
```

In order to run this program, you must first create a resource file called MENU2.RSC. If you have a resource construction program, you'll need to create an object tree, an alert, and two free strings. The object tree is is a menu tree with the C macro name MENUTREE. This menu has three titles. To the default DESK and FILE titles add an OPTION title. Under the DESK menu, edit the first item's string to read About Menu2 . . . and give it the C macro name ABOTITEM. Edit the item under the FILE menu to read Quit ^Q and give it the name QUITITEM. Next, add four items under the OPTION menu. The first reads Check Mark and has the name CHEKITEM. The second is just a gray separator bar. The third reads *Turn the Next Item ON* and has the name TOGLITEM. The fourth is a disabled sting that reads *Print Message* in faint letters. The alert string called called ABTALERT has four string lines. The first reads *Menu demo with check marks,* the second reads *graying, and alternate text,* and the third reads *Select "Quit" to end.* The fourth line is made up of dashes. The string in the button should be edited to read *I'll remember that.* You'll also have to create two free strings. The first is called OFFSTRNG and reads *Turn Next Item OFF,* while the second is called ONSTRNG and reads *Turn Next Item ON.* If you don't have a resource construction program, you'll be able to build the resource file needed for this example by running the RSCBUILD.C program in Appendix C.

Chapter 6

Interactive Object Handling: Forms and the File Selector

The highest level of AES object routines are so sophisticated that they're more like large subprograms than simple functions. The programs in the Forms and File Selector Library perform all the same functions as the sample object programs from Chapter 4, and more. They display a set of objects (called *a form*) which is loaded as one of the object trees in the resource file. They check for significant AES events and handle all of the user's interaction with these objects. Finally, when they've detected an event that ends the interaction, they report its results to the application. This means that the programmer can easily create interactive forms with his or her resource construction program and can let these functions take care of the the job of watching for user input. The AES aids the user in filling in the blanks and checking the boxes and provides the results to the application.

Forms

The simplest kind of form is an error box. This is a box object which contains an image, a text string, and a button. The image, which appears at the left side of the box, is that of a stop sign. The text string, which appears at the top of the box, either contains an explanatory error message or just reads *TOS error* #X, where X is a number that corresponds to a TOS system error. Finally, the button appearing at the bottom of the box reads *Cancel.* This form makes it easy to let the user know when a TOS error occurs. When a TOS function returns an error number, your program can simply call *form_error( ),* which will display the error message box until the user clicks on the Cancel button. The syntax for the form_error call is

```
int exitbutn, error;
exitbutn = form_error(error);
```

where *error* is the error number. Unfortunately, GEM is designed with IBM PC-DOS error codes in mind, rather than GEMDOS error codes returned on the Atari ST computers.

Therefore, in order for form_error( ) to print out its error messages correctly, you must convert the GEMDOS codes to PC DOS codes. This is done by reversing the sign of the code from negative to positive and then subtracting 31 (DOS_ERR = (−TOS_ERR) − 31). The following chart lists the GEMDOS error codes for which form_error( ) prints error-specific messages (as opposed to *TOS error #X*) and gives the complete text of those messages.

GEMDOS Error Number	PC-DOS Error Number	Error	Error form_error( ) Message
−33	2	File not found	This application can't find the folder or file you just tried to access.
−34	3	Path not found	This application can't find the folder or file you just tried to access.
−35	4	File-handle pool exhausted (no file handles left)	This application doesn't have room to open another document. To make room, close any document that you don't need.
−36	5	Access denied (wrong attribute or access code)	An item with this name already exists in the directory, or this item is set to Read-only status.
−39	8	Insufficient memory	There isn't enough memory in your computer for the application you just tried to run.
−41	10	Invalid environment	There isn't enough memory in your computer for the application you just tried to run.
−42	11	Invalid format	There isn't enough memory in your computer for the application you just tried to run.
−46	15	Invalid drive specification	The drive you specified does not exist. Check the drive's identifier or change the drive identifier in the DISK INFORMATION dialog.

GEMDOS Error Number	PC-DOS Error Number	Error	Error form_error( ) Message
−47	16	Attempted to remove the current directory	You cannot delete the folder in which you are working.
−49	18	No more files	This application can't find the folder or file you just tried to access.

Although GEM provides codes for more than one exit button, the current ST version uses only the Cancel button, so that a value of 0 is always returned in *exitbutn*.

Alerts

The error form is actually a specialized case of a more versatile message display form called the *alert*. Alert boxes are used to inform the user that a situation has arisen in which some immediate action may be required. Alerts make it easy to present to the user a short message and a choice of up to three options. A typical use is verifying that the user really wants to take some irrevocable action, such as *Loading new data will destroy current data. Do you wish to proceed? [Yes] [No]*. Another use of the alert would be to inform the user of an error as in *Data file cannot be loaded [Cancel]*.

The alert box consists of an optional image, up to five lines of text, and from one to three exit buttons. To display the box, you use the *form_alert( )* call, whose syntax is

```
int exitbutn, default;
char *string;
exitbutn = form_alert(default, string);
```

where *default* is the number of the default exit button. Since there may be up to three exit buttons with which the user may close the alert box, the default should be a number in the range 1–3. The variable *string* is a pointer to a specially formatted text string which describes the image (if any), the message text, and the exit buttons. The format for this string is

```
"[Icon_number][Message text][Exit button text]"
```

This string is separated into three parts, each of which is set off by square brackets. The first item, icon_number, is a single numeric digit indicating which image (if any) should be displayed at the left side of the alert box. The choices are

Image Number	Image	Meaning
0	None	
1	Exclamation point in diamond	Note
2	Question mark in yield-sign triangle	Wait
3	Octagonal stop sign	Stop

These images are used to indicate alerts of increasing importance. The NOTE alert may be used to pass information that's only of casual interest to the user, while the STOP alert should be reserved for the situation where data may be lost if the user proceeds.

The second set of square brackets holds the text message. This message is limited to a maximum of five lines, each of which may contain a maximum of 40 characters. The vertical bar character (|) is used to indicate the start of a new line. For example, the string [This is line 1 | This is line 2 | This is line 3] prints in an alert as

This is line 1
This is line 2
This is line 3

The final set of square brackets contain the text for the exit buttons. A maximum of three exit buttons may be used, each of which contains a maximum of 20 characters of text. As with the message string, the text for each button is separated with a vertical bar character. When the alert concludes, the number of the exit button (1, 2, or 3) is returned in the *exitbutn* variable. Note also that when the alert ends, the AES automatically restores the screen rectangle it had covered. Since the AES stores the screen background in the menu/alert buffer, it can use the raster *blit* functions to replace it without intervention from your program.

There are two ways to generate an alert string. The first is to create it using a resource construction program, save it as part of the resource file, and then use the *rsrc_gaddr( )* function with the R_STRING type to find its address. The menu program examples in the preceding chapter use this method to display the alert boxes for the About . . . menu items. The second way is to simply include the string constant as part of your form_alert call, like this:

```
form_alert(1, "[2][ To be or not to be . . . | That's a question? |
][YES | NO | MAYBE]");
```

This method is used in the menu examples to generate the alert that notifies the user that the resource file can't be loaded, since you can't very well get this string from the resource file.

Dialog Boxes

The final type of form, the dialog box, is much more flexible than simple error boxes or alerts. Dialog boxes may contain any number of GEM objects of any type. They may be used to present a large number of on-off selection buttons, along with mutually exclusive radio buttons. They may include neatly formatted text strings into which the user may enter information via the keyboard. And, as you will see, with a little work they can even contain more sophisticated constructs such as slide bars.

The Form Library routine *form_do( )* is used to animate a dialog once it's been displayed. In most cases, form_do( ) handles all the user's interaction with the objects in the dialog. For example, if there are any formatted text objects in the dialog (G_FTEXT or G_FBOXTEXT), form_do( ) positions the text cursor at the first editable field and handles all the keyboard input from the user. If the user enters a valid text character (one that matches the criteria of the validation string), form_do( ) inserts it into the text string. If the user types an invalid character, form_do( ) ignores that character. If the user types an invalid character that's part of the template, form_do( ) moves the cursor to the first space past that template character. For example, if the template string is Date:__/__/__, when the user enters a slash, the cursor moves to the space following the next slash in the template.

Form_do( ) also handles a number of cursor and control keystrokes. These include the cursor keys, Tab, Shift-Tab, Delete, Backspace, Esc, and Return.

Left and right cursor keys. These move the text cursor backwards or forwards through the text field.

Up and down cursor keys, Tab, and Shift-Tab. The down-arrow key or Tab key can be used to move the text cursor to the next editable text field. The up-arrow key or Shift-Tab combination can be used to move to the previous text field. The cursor moves to the first open character position in the text field.

Delete and backspace. Delete removes the character to the right of the cursor, while the backspace key removes the character to the left of the cursor.

Esc. The Esc key clears all characters from the text field.

Return. Return selects the first object with the DEFAULT flag set in its *ob_flags* field. This object is highlighted, and the form_do( ) ends, returning the object number of the DEFAULT object. If no objects are designated as DEFAULT, form_do( ) ignores the Return key.

The other major task that form_do( ) performs is handling selection of objects that have the SELECTABLE flag set in their ob_flags field. These objects may be of the type G_BOX, G_BOXTEXT, G_BUTTON, or even G_IMAGE or G_ICON.

To select such an object, the user moves the mouse pointer over the object, presses the left mouse button, then releases it. Form_do( ) checks to see which object the pointer is over when the user clicks and selects that object. When an object is selected, it is redrawn in its highlighted form. This usually means that the entire object rectangle is inverted, so that each pixel of color is changed to its complement. The program may, however, use its own form of highlighting. It can do this either by using TOUCHEXIT objects so that the program regains control and can do its own drawing when the user clicks on them, or by using G_PROGDEF type objects so that the program's own routine is called automatically when the object is to be drawn or highlighted.

As part of its object selection service, form_do( ) handles the mutual-exclude feature of objects whose flag settings include RBUTTON in their ob_flags field. As explained in Chapter 4, these objects, which must all be siblings, deselect all others when one is selected. Form_do( ) respects this flag and makes sure that only one radio button is selected at a time.

Form_do( ) will continue handling input from the user until an exit condition occurs. The most common exit condition occurs when the user selects an object that has the EXIT and SELECTABLE flags set in its ob_flags field. Typically, exit objects include buttons that read OK or Cancel. If one of these objects has its DEFAULT flag set, then the user can exit form_do( ) by pressing the Return key.

Finally, if an object has the TOUCHEXIT flag set, form_do( ) will exit as soon as the user moves the mouse pointer to the object and presses the button down. This allows the programmer to create draggable object types by seizing control when the user starts to drag the object. When form_do( ) exits, it returns the object number of the object whose selection terminated the form_do( ) call.

There is a set procedure to follow when using form_do( ) to animate a dialog. First, load the resource file with the *rsrc_load()* call. Next, find the address of the dialog tree with the *rsrc_gaddr()* call. Then, use the *form_center()* command to center the dialog box on screen. The syntax for this call is

```
int x, y, width, height;
OBJECT *tree;
reserved = form_center(tree, &x, &y, &width, &height);
```

where *tree* is a pointer to the dialog object tree, and *x*, *y*, *width*, and *height* are the variables in which the routine returns the position and size of the centered dialog box. This routine computes the top left coordinates at which the dialog box will be centered on screen and writes those coordinates into the *ob_x* and *ob_y* fields of the root object. This step is necessary because the root objects created by the resource construction programs are all positioned at (0,0). If you don't center these objects before displaying them, dialog boxes will always be drawn in the top left corner of the screen, rather than in the middle.

The next step is to use the *form_dial()* call to reserve the screen area in which the dialog is to be displayed. The syntax for this call is

```
int status, action, smallx, smally, smallw, smallh;
int largex, largey, largew, largeh;
status = form_dial(action, smallx, smally, smallw, smallh, largex,
largey, largew, largeh);
```

where *action* is a flag which indicates the type of action you wish to take, and the two sets of rectangle information give the smallest and largest dimensions of the dialog box. The four valid action-type flags (and the macro names for them defined in the GEMDEFS.H file) are as follows:

Type Number	Macro Name	Action
0	FMD_START	Reserves the screen area used by the dialog box
1	FMD_GROW	Draws expanding box from small to large rectangle
2	FMD_SHRINK	Draws shrinking box from large to small rectangle
3	FMD_FINISH	Frees the screen area used by the dialog box, and causes redraw messages to be sent

The type of action that is appropriate at this stage of the dialog presentation is FMD_START, which reserves the screen area that will be used by the dialog box. If you've read the official GEM documentation, you may have noticed that the syntax for the C binding shown there includes only one set of rectangle parameters. The actual binding, however, uses the format shown above. Although only the two middle action types—FMD_GROW and FMD_SHRINK—use both sets of rectangles, you must supply dummy parameters as place holders when you use the other two action types. For example, a typical FMD_START command would take the form

form_dial(FMD_START, 0, 0, 0, 0, x, y, width, height);

The next step is completely optional. If you wish, you may call form_dial using the FMD_GROW subcommand. This step animates a *zoom box* which moves and grows from the first rectangle to the second one. While this may add a bit of visual appeal to your program, it also causes a slight delay which may irritate the more advanced user. Generally, it's best not to include it unless it gives the user some meaningful information. For example, if the user double-clicks on an icon to open a dialog box, then it may be helpful to show the dialog box "exploding" from that icon to show the relationship between the two. If the dialog box was started up from a menu item, however, it really doesn't add anything to show the box exploding from the menu since the menu item disappears before you can tell where the zoom box is coming from.

Finally, you draw the dialog box using the *objc_draw()* command and animate it using *form_do()*. The syntax for the form_do( ) call is

```
int exitobj, editobj;
OBJECT *tree;
exitobj = form_do(tree, editobj);
```

where *tree* is a pointer to the dialog's object tree and *editobj* is the object number of the editable text object at which the cursor will first be placed. If there are no editable text fields, you should pass a value of 0 for *editobj*. The *exitobj* variable contains the number of the object whose selection caused the end of the dialog animation. If that object had the TOUCHEXIT flag set and was selected with a double-click, the high bit of the exitobj field will be set. If you use TOUCHEXIT objects and don't care to check for double-clicks, you may want to mask that bit off before checking the object number.

When form_do( ) terminates, the dialog is still displayed onscreen. If the dialog has truly concluded, it's up to your application to clean up the screen. If you had used to FMD_GROW subcommand of form_dial( ) at the beginning of the dialog, you'll want to use the FMD_SHRINK subcommand of form_dial( ) to reverse the zoom box.

After that (optional) step, you'll want to use the FMD_FINISH subcommand of form_dial( ) to release the screen area that had been reserved for the dialog and to restore the screen. Since a dialog box can be considerably larger in size than one quarter of the screen, it's impossible to store the screen background in the menu/alert buffer. Therefore, the AES cannot automatically restore the screen background at the end of a dialog. Instead, it considers the screen rectangle described by the form_dial(FMD_FINISH) call as damaged and sends redraw messages to all of the windows within this area. If your program handles those messages already, then the redraw will be more or less automatic.

You should make certain that the screen rectangle that you describe in the form_dial(FMD_FINISH) call really covers all of your dialog box. The *x*, *y*, width, and height values returned by form_center( ) allow for the OUTLINED state, which is the default for the root object of dialog boxes created with the resource construction programs. But it doesn't take outside borders into consideration. If you have a thick exterior border, you may need to extend the size of the rectangle you describe in form_dial(FMD_FINISH). Note, however, that form_dial(FMD_FINISH) always sends redraw messages for an area two pixels wider and higher than the one specified, which means that you don't have to allow for the drop shadow under SHADOWED objects.

Although form_dial(FMD_FINISH) was designed for dialog handling, you may use it any time you wish to force a redraw. This method is much handier than using the message pipe to send a redraw message directly, since the AES figures out which windows should get the message and sends them automatically. In addition, this call forces the AES to redraw the Desktop window in those areas not covered by application windows. Since more work is involved, this method is a bit slower than sending the redraw messages directly.

Once your program has exited the dialog, there are still some tasks left to perform. First, you should deselect the exit button. You can do this either by using the *objc_change()* routine or by directly resetting the SELECTED bit of the objects *ob_state* field. If you fail to do this, the next time the dialog is displayed, it will come up with the object used to exit the dialog highlighted in inverse video.

You must also note new selection settings for all the significant objects in the dialog. That includes any text that may have been entered into editable text objects. You should probably transfer all the new settings to a separate array which keeps track of current settings. That way, your program won't have to keep checking the object fields to find the settings. Also, this practice will allow you to undo changes the user makes if he or she exits the dialog with the Cancel button instead of OK.

Program 6-1 demonstrates some of the common features of dialog-box handling.

Program 6-1. dialog1.c

```
/************************************************************************/
/*                                                                      */
/*      DIALOG1.C -- Demonstrates handling of simple dialog             */
/*      box, with buttons, boxes and editable text.                     */
/*                                                                      */
/************************************************************************/

#define APP_INFO " "
#define APP_NAME "Dialog Example 1"
#define WDW_CTRLS (NAME)
```

```
#define MOUSE_OFF graf_mouse(M_OFF,ØL)
#define MOUSE_ON  graf_mouse(M_ON,ØL)
#define FALSE Ø
#define TRUE 1
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )

#include "aesshell.c"
#include "dialog1.h"   /* include file from RCS */


OBJECT *menutree;   /* pointer to menu object tree */
OBJECT *dialtree;   /* pointer to dialog object tree */
TEDINFO *obspec;    /* pointer to TEDINFO for text object */
char *strptr;       /* pointer to text string for text object */

int ages[3]= {Ø, Ø, Ø};          /* array for age tally */
int computers[4]= {Ø,Ø,Ø,Ø};     /* array for computer tally */
char lastxt[13]= "";             /* buffer for last "other" text */
int left, top;

demo()
{

int done = Ø;
int msg[8];

if (!rsrc_load("DIALOG1.RSC"))       /* Load resource file */
   {
   form_alert(3,"[Ø][Fatal Error!Can't find DIALOG1.RSC file!][Abort]");
   return(Ø);                      /* Abort if it's not there */
   }

/* get address of menu tree and dialog tree */
rsrc_gaddr(R_TREE, MENUTREE, &menutree);
rsrc_gaddr(R_TREE, DIALTREE, &dialtree);
/* get pointer to text objects's text string */
obspec = (TEDINFO *)dialtree[OTHERTXT].ob_spec;
strptr = obspec->te_ptext;  /* pointer to text string */
strptr[Ø] = Ø;              /* clear out string */

MOUSE_OFF;                   /* Hide the mouse pointer */
menu_bar(menutree,1);        /* Show the menu bar */
MOUSE_ON;                    /* Show the mouse pointer */

/* Main Program Loop */

   while (! done)     /* until user selects "Quit" item */
   {
   evnt_mesage(msg);  /* check menus */
   done = handle_msg(msg);  /* & handle messages */
   } /* end of main WHILE loop */

menu_bar(menutree,Ø);       /* Remove the menu bar */

} /* end of DEMO function */

/* Message Handler routine -- only handles menu messages */
/* and redraws (yours should also handle window topping, etc.) */

handle_msg(msg)
int msg[8];
{

   int done=Ø;
```

```
   switch (msg[Ø])         /* check message type */
   {
   case WM_REDRAW:         /* if redraw, call refresh routine */
      refresh(msg[3], (GRECT *)&msg[4]);
      break;
   case MN_SELECTED:       /* if menu message type */
      switch (msg[4])      /* check menu item */
      {
      case SURVITEM:       /* Do Survey dialog */
         do_dial();
         break;
      case QUITITEM:       /* Quit */
         done = 1;
         break;
      default:
         break;
      } /* end of switch on menu item */
      menu_tnormal(menutree, msg[3], 1); /* set menu to normal */
      break;

   default:
      break;
   }/* end of switch on message type */

   return(done);           /* report done status */
}

/* Dialog Handler routine -- Displays survey form */

do_dial()
{
   int x, y, width, height, exitbutn;

   form_center(dialtree, &x, &y, &width, &height);
   form_dial(FMD_START, Ø, Ø, Ø, Ø, x, y, width, height);
   objc_draw(dialtree, ROOT, MAX_DEPTH, x, y, width, height);
   exitbutn = form_do(dialtree, OTHERTXT);
   check_objs(exitbutn); /* check object states, etc. */

   form_dial(FMD_FINISH, Ø, Ø, Ø, Ø, x, y, width, height);
}

/******** Sub-function to check dialog objects ********/
check_objs(exitbutn)
   int exitbutn;

{
   int x;

   dialtree[exitbutn].ob_state ^= SELECTED; /* de-select exit button */

   for (x=YUNGBUTN; x<(OLDBUTN+1);x++)       /* check radio buttons */
      {
      if (dialtree[x].ob_state & SELECTED)
         {
         if (exitbutn==OKBUTN) ages[x-YUNGBUTN]++; /* increment age */
         dialtree[x].ob_state ^= SELECTED;         /* reset buttons */
         }
      }

   for (x=STBOX; x<(OTHERBOX+1);x++)         /* check select boxes */
      {
      if (dialtree[x].ob_state & SELECTED)
         {
         if (exitbutn==OKBUTN) computers[x-STBOX]++; /* increment computers
         dialtree[x].ob_state ^= SELECTED;           /* reset buttons */
         }
      }
```

```
   if ( (strlen(strptr)>0) && (exitbutn == OKBUTN) )
      strcpy(lastxt, strptr);     /* if string not empty, copy */
       strptr[0]=0;               /* clear object string */

}


/******* Sub function to handle window refresh ***************/

refresh(wh, drect)   /* routine to handle window_refresh (WM_REDRAW) */
   int   wh;         /*  window handle from msg[3] */
   GRECT *drect;     /*  pointer to damage recatangle  */
   {
   GRECT   wrect;    /*  the current window rectangle in rect list */

   MOUSE_OFF;        /* turn off mouse */
   wind_update(BEG_UPDATE);    /* lock screen */

   wind_get                          /* get first rectangle */
    (wh, WF_FIRSTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w, &wrect.g_h);

   while ( wrect.g_w && wrect.g_h ) /* while not at last rectangle,  */
      {
      if (overlap(drect, &wrect))   /* check to see if this one's damaged, */
         {
            set_clip(&wrect);       /* if it is, set clip rectangle */
            display(wh);            /* redraw, and turn clip off */
            vs_clip(handle, FALSE, (int *)&wrect );
         }
      wind_get(wh, WF_NEXTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w,
         &wrect.g_h);
      }

   wind_update(END_UPDATE);          /* unlock screen */
   MOUSE_ON;                         /* turn mouse pointer back on */
   }


/******** Subfunction to draw the window display **********/

display(wh)          /* draw the window display */
  int wh;
{
  char buf[80];
  int x, y, width, height;

  wind_get(wh, WF_WORKXYWH, &work.g_x, &work.g_y,
          &work.g_w, &work.g_h);    /* find work area */
  clear_rect(&work);                /* and clear it */

   form_center(dialtree, &x, &y, &width, &height);
   left = x + cellw;
   top = y + cellh;


  sprintf(buf, "Computers ");
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "------------------ ");
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "Atari ST's = %d% ",computers[0]);
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "Atari XL/XE's = %d% ",computers[1]);
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "Exidy Sorceror's = %d% ",computers[2]);
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "Other = %d% ",computers[3]);
  v_gtext(handle, left, top+=cellh, buf);
  sprintf(buf, "Others include: %s% ",lastxt);
  v_gtext(handle, left, top+=cellh, buf);
```

```
    sprintf(buf, "Ages ");
    v_gtext(handle, left, top+=(cellh*2), buf);
    sprintf(buf, "\----------------- ");
    v_gtext(handle, left, top+=cellh, buf);
    sprintf(buf, "Under 16 = %d% ",ages[0]);
    v_gtext(handle, left, top+=cellh, buf);
    sprintf(buf, "16-39    = %d% ",ages[1]);
    v_gtext(handle, left, top+=cellh, buf);
    sprintf(buf, "Over 39  = %d% ",ages[2]);
    v_gtext(handle, left, top+=cellh, buf);

}

/* >>>>>>>> Utility routines used by other functions <<<<<<<<<<<<< */


set_clip(r)   /* set clip to specified rectangle  */
   GRECT   *r;
   {
   int  points[4];

   grect_conv(r, points);
   vs_clip(handle, TRUE, points);
   }


overlap(r1, r2)      /* compute overlap of two rectangles   */
   GRECT   *r1, *r2;
   {
   int x, y;

   x = MAX(r2->g_x, r1->g_x);
   y = MAX(r2->g_y, r1->g_y);
   r2->g_w = MIN(r2->g_x + r2->g_w, r1->g_x + r1->g_w) -x;
   r2->g_h = MIN(r2->g_y + r2->g_h, r1->g_y + r1->g_h) -y;
   r2->g_x = x;
   r2->g_y = y;
   return( (r2->g_w > 0) && (r2->g_h > 0) );
   }


/* ****************** End of Dialog1.c ************** */
```

In order to run this program, it's first necessary to create the resource file DIALOG1.RSC. This file contains two object trees, a dialog tree called DIALTREE, and a menu tree called MENUTREE. The dialog tree sets up a form for a computer survey. On the top line is a G_STRING that reads *COMPUTER SURVEY.* On the next line is a G_STRING that reads *Age:*. Next to the G_STRING is a borderless G_IBOX that covers the rest of the line. Within that box are three radio buttons, named YUNGBUTN, MIDBUTN, and OLDBUTN. Each of these buttons have the SELECTABLE and RBUTTON flags set. The text of YUNGBUTN reads *Under 16,* the text of MIDBUTN reads *16–39,* and the text of OLDBUTN reads *Over 39.* On the next line of text down is a G_STRING that reads

Computers Owned:. Under that string appears four G_BOXes, one on top of the other. Their names are STBOX, XLBOX, EXIDYBOX and OTHERBOX. Next to the top three boxes are G_STRINGS that read Atari ST, Atari XL/XE, and Exidy Sorcerer. The fourth box, OTHERBOX, has a G_FTEXT next to it called OTHERTXT. The template for this text is *Other:* followed by 12 underscore characters. The validation string is 12 *n* characters, and the text is 12 spaces.

The menu tree has the default DESK and FILE titles with one menu item added to the FILE menu. Along with the QUIT item (whose name is QUITITEM), there's an item that reads Survey . . . whose name is SURVITEM. Using any of the resource construction programs, it should be fairly easy to create the resource described here. If you don't have a resource construction program, you'll need to create the resource file from data arrays as shown in Appendix C.

There are several interesting points to note about this program. First, note how the text of the editable string was reset by assigning a pointer to this string (named strptr), and then setting the first character of the string to ASCII 0 with the statement *strptr[0] = 0;*. This insures that when the dialog comes up, the editable text string will be blank, and the cursor will be positioned at the first character of the string. Before you clear out the string, however, copy the last string entered into the *lastxt* array, using the C function *strcpy( )*.

Another important point to note is how the program takes advantage of the object tree structure in tallying the votes from the form. Since the objects YUNGBUTN, MIDBUTN, and OLDBUTN are contiguous in the object array, it's possible to check their SELECTED flags by using a loop that runs from object YUNGBUTN to object OLDBUTN. The same is true of the G_BOX objects STBOX, XLBOX, EXIDYBOX, and OTHERBOX. You should create these objects one after the other with the resource construction program, in order to insure that they have consecutive object numbers.

Although dialog boxes are an easy way to receive input from the user, they do have some drawbacks. For one thing, there are some bugs in the current (preblitter) implementation. For example, if the user enters the underscore character in an editable text field, the machine crashes immediately. Although Atari has promised to fix this rather serious flaw in the next version of the operating system, programmers might be wary of using this potentially disastrous routine in their programs.

The most serious problem with form_do( ) is that the routine takes complete control of the program until the user hits the exit button. That means that while form_do( ) is executing, the user is stuck with providing input to the form and can't do anything else. The menu system doesn't work while a dialog is on screen, and if the user clicks on an object outside of the dialog box, all that happens is that a bell sounds. The reason for this is that form_do( ) itself uses the *evnt_multi( )* call and only checks for keyboard and mouse button events. The evnt_multi( ) call takes place in a loop that keeps repeating until an exit condition occurs.

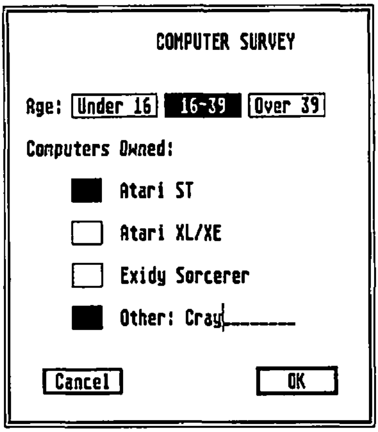
Many programmers consciously try to avoid program *modes*. Form_do is modal by its very nature, since it puts the user into a form input mode, where he can do nothing but enter information, and where normal program features like menus don't work. Programmers who wish to design "modeless" programs, which allow the user to click on objects, enter text, and use menus all at the same time, must therefore write their own form_do( ) equivalent, integrating its evnt_multi call into the program's main evnt_multi loop. Tim Oren, a member of the original GEM design team, has posted an excellent example of a user-defined form_do( ) routine in the Utilities Data Library (DL 3) area of the Atari16 Forum on the CompuServe Information Service. The source code can be found in the file named GMCL13.C.

The File Selector

The final type of interactive form which the AES provides is the File Selector. The Library routine *fsel_input( )* displays a complete prefabricated dialog box. The purpose of this dialog is to provide a standard method for showing the user the contents of a disk and allowing the selection of one of the files on that disk. The dialog box displayed by fsel_input( ) is shown in Figure 6-1.

The current search path (drive name and subdirectories, if any) is shown on the line beneath the title Directory. The directories and files found there are displayed in the window which is located in the lower left corner of the dialog box. A solid block with a white diamond shape in it appears before the names of directories, to distinguish them from files. There is a scroll bar at the side of the window, which may be used

Figure 6-1. Dialog Box Displayed from the Program dialog1.c



to view additional names if there are more than can fit in the box. If the user wishes to view a different drive or directory, he or she may change the display by clicking on the Directory blank and typing in a new file specification such as *A:* or *C:\PROGRAMS*. After typing in the new path, the user clicks anywhere within the filename window, and the AES updates its contents according to the new specification. If the user wishes to see the contents of any subdirectory whose name is in the window, he or she only has to click on the directory name, and the Directory specification will automatically be changed to show the contents of that subdirectory.

A file may be selected in several ways. The user may type in the name of the file in the space marked Selection and then click on the OK button or press Return. A second way to select the file would be to click on the name of the file in the window, which will cause that filename to appear in the Selection blank, and then click on the OK button or press Return. Another way would be to double-click on the name of the file, which will both select it and choose OK.

To display the File Selector dialog, you use the *fsel_input()* call, whose syntax is as follows:

```
int status, exitbutn;
char path[64], file[13];
status = fsel_input(path, file, &exitbutn);
```

where *path* is a pointer to a string which contains the initial pathname and *file* is a pointer to a string which contains the default filename. The path string contains the drive letter, the subdirectory path, and the filename using a wild card, such as *C:\WORDPROC\L+ETTERS*.DOC.* The filename can be used to specify a default file or may be left blank. These same two variables are also used by fsel_input( ) to return the pathnames and filenames selected by the user. If the Cancel button was selected, however, the original values will be unchanged upon exit from this routine. The *exitbutn* variable is where the routine returns the number of the button used to exit the dialog (0 = Cancel, 1 = OK).

The screen display isn't automatically restored upon completion of the fsel_input( ) call. That's because the dialog box is larger in size than one quarter of the screen, so the screen background can't be saved in the menu/alert buffer. Therefore, when the function ends, the AES determines that the dialog rectangle is "dirty," and sends a WM_REDRAW to every application whose windows lay within the rectangle. If your program is prepared to handle WM_REDRAW messages, the display will be cleaned up as part of its routine message-handling chores. Note, however, that fsel_input( ) changes the current VDI clipping rectangle and doesn't change it back upon exit from the routine. Therefore, if your program does any VDI rendering after a call to fsel_input( ), you'll probably have to set the clipping rectangle afterward, whether you normally use clipping or not.

Program 6-2 demonstrates use of the fsel_input( ) routine to get a filename from the user. For purposes of brevity, this program simply clears the screen when it gets the WM_REFRESH message. In your own program, you would want to include an entire message-handling system that would take care of menu messages, window topping messages, refresh messages, and so on.

Program 6-2. fselect.c

```
/*********************************************************************/
/*                                                                   */
/*      FSELECT.C -- Demonstrates the use of the File               */
/*      Selector routine to obtain the pathname of a file.          */
/*                                                                   */
/*                                                                   */
/*********************************************************************/

#define APP_INFO " ** Click on the Close Box to exit the program. **"
#define APP_NAME "File Selector Demo"
#define WDW_CTRLS (NAME|CLOSER|INFO)
```

```
#include "aesshell.c"

char file[64];        /* buffer for name of file selected */
char path[64];        /* buffer for search path */
char filespec[80];    /* buffer for full pathname of file */

demo()
{
  int msg[8];
  int button, z;

    path[0]=Dgetdrv()+'A';   /* Get the drive name */
    path[1]=':';

    Dgetpath(file,0);        /* and current directory path */
    strcat(path,file);

    file[0] = '\0';              /* no initial file name */
    strcat(path,"\\*.*");        /* set initial path */
    fsel_input(path, file, &button);  /* get file name */

    evnt_mesage(&msg);  /* get the window redraw message, */
                        /* and handle it (sort of) */
    graf_mouse(M_OFF, 0L);      /* turn the mouse pointer off */
    clear_rect(&work);          /* clear the area */
    graf_mouse(M_ON, 0L);       /* and turn it back on */

    strcpy (filespec,path);  /* copy path to filespec buffer */
    z = strlen(filespec);    /* remove characters from end... */
    while(z && (filespec[z-1] != '\\') )
       z--;                  /* until you get to backslash */
    filespec[z]='\0';
    strcat(filespec,file);   /* and add filename to path */

/* print complete path/filename */
    v_gtext(handle,10*cellw, 10*cellh, filespec);

    evnt_mesage(&msg);  /* & wait for the window close message */

}

/********************** End of Fselect.c ************************/
```

Although fsel_input( ) takes care of all of the interaction with the user, some additional steps are required before and after making the call. For one thing, the user may have put your program on a second floppy or hard drive, so it might be frustrating to find it if you always used the A drive for the default search path. The safest course of action is to set the default search path to the current directory, the one from which the program was run. In order to find the current directory, it's necessary to use a couple of GEMDOS routines. The first routine is referred to by the macro name *Dgetdrv* which is defined in the OSBIND.H file. This call returns the current drive number (0 = A, 1 = B, and so on). The second routine is called *Dgetpath*. Given a pointer to a 64-byte buffer and a drive number (0 = current drive, 1 = A), this routine places a string which specifies the directory path in the buffer. In the example program above, these two routines are used to build the default pathname.

Another bit of manipulation is needed after fsel_input( ) returns the pathname and filename chosen. Since the path specifier will usually contain a wildcard for the filename, you have to eliminate each character from the end of the path string until you come to the backslash character. You can then tack on the filename to the end of the pathname.

Although the File Selector does provide a uniform method for obtaining a filename from the user, it has some limitations. Chief among these is the fact that the filename window will only hold 100 entries. This means that if there are more than 100 files in a directory, the user won't be able to view the last ones. Also, the file selector doesn't give the user any feedback on which drives are currently available. For these reasons, some programmers prefer to use their own file-selection dialogs. If you are planning to do so, however, it would probably be a good idea to keep the form and function of this dialog as close to the standard one as possible.

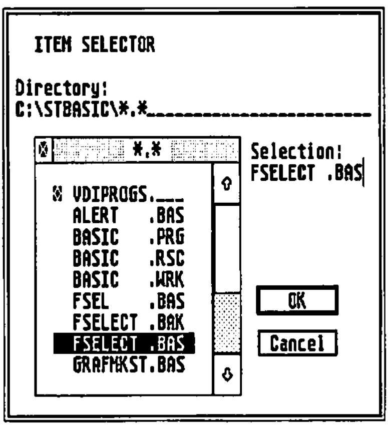
The fsel_input( ) routine is one that can be profitably used from a BASIC program. Program 6-3 demonstrates how to use this call from BASIC. Note that in this version of the program, the default path was set to *A:*.**, since we are unable make the necessary GEMDOS calls to determine the current path from the first version of STBASIC. Figure 6-2 illustrates the file selector dialog box.

Program 6-3. fselect.bas

```
10    '*****************************************
20    '* FSELECT.BAS -- Demonstrates use of the *
30    '* File Selector Library in ST BASIC      *
40    '*****************************************
50    apb# = gb
60    CONTROL = peek(apb#)
70    GLOBAL  = peek(apb#+4)
80    GINTIN = peek(apb#+8)
90    GINTOUT = peek(apb#+12)
100   ADDRIN# = peek(apb#+16)
110   ADROUT# = peek(apb#+20)
120   fullw 2
130   PATH$="A:\*.*"  'Set pathname, and expand string to 64 characters
140   PATHNAME$=PATH$+string$(58,chr$(0))
150   FILE$=""+string$(12,chr$(0))  'Set filename to 13 nuls
160   poke ADDRIN#,varptr(PATHNAME$)
170   poke ADDRIN#+4,varptr(FILE$)
180   gemsys(90)  'Call fsel_input()
190   exitbutn = peek(GINTOUT+2)
200   X = 1 'Truncate file name
210   while (ASC(mid$(FILE$,X,1))<>0)
```

```
220   X=X+1
230   wend
240   FILE$=left$(FILE$,X)
250   X= len(PATHNAME$) 'Truncate path name
260   while(mid$(PATHNAME$,X,1)<>"\")
270   X = X-1
280   wend
290   PATHNAME$ = left$(PATHNAME$,X)
300   clearw 2:gotoxy 10,10
310   print PATHNAME$+FILE$
320   gotoxy 10,12:? "Press any key to end...":X=inp(2)
330   clearw 2: end
```

Figure 6-2. The File Selector Dialog Box



Chapter 7

The Graphics Library

Most of the graphics routines that GEM uses are found in the VDI graphics library. However, a few higher-level graphics routines were needed in order to implement the complicated object-manipulation routines of the Forms Library and some of the features of the GEM Desktop applications.

These higher-level routines are found in the AES Graphics Library. These functions may appear to be so specialized that their usefulness is not readily apparent. When examining them, therefore, you should focus on the ways these routines are used by the GEM Desktop program. Then you should consider similar ways you might use them in your programs.

Box Manipulation

Many of the AES graphics routines are used to manipulate the rectangular outline of a box drawn on the screen. If you stop to think, you'll realize how much is done on just the Desktop program with boxes. Rectangular shapes are used for windows, icons, buttons, and slider bars. These shapes are moved, dragged, and sized with the help of the AES graphics routines.

The first of these routines is used to draw a *rubber box.* This is a box composed of dotted lines. The upper left corner of the box remains fixed, but the lower right corner changes position as the user drags the mouse. As long as the user holds the left mouse button, the box is redrawn whenever the mouse pointer changes positions. When the button is released, the box is erased. The operation of the rubber-box routine should be familiar to anyone who has used the size box on a window. The name of the routine used to draw the rubber box is *graf_rubberbox( ).* The syntax for this call is

```
int status, x, y, minw, minh, endw, endh
status = graf_rubberbox(x, y, minw, minh, &endw, &endh);
```

where *x* is the *x* coordinate for the left side of the box, and *y* is the *y* coordinate of the top of the box. The variables *minw* and *minh* contain the minimum box width and height, respectively. When the function ends, the current box width and height are returned in *endw* and *endh*. The function's error status is returned in the status variable. A status of 0 signals that an error occurred during the call, while a positive integer means that function completed without error.

Typically, a program will only call this routine when it has determined that the mouse button is down. This may be determined by an *evnt_multi()* or *evnt_button()* call, or by the activation of an object whose TOUCHEXIT flag is set. When the call is made, the dotted outline of a box is drawn from the coordinates *leftx* and *topy* to the current mouse position. The function continues to track the mouse pointer until the user releases the left mouse button.

The *graf_dragbox()* function allows the user to drag a box within a boundary rectangle, such as a window or dialog box. As with graf_rubberbox( ), this function should not be started until the program has determined that the user has pressed the left mouse button down. When the function is called, it draws the box outline and redraws the box whenever the mouse pointer moves within the boundary rectangle, until the left mouse button is released. When the function ends, it erases the box outline and returns the ending position of the box. The syntax for the graf_dragbox( ) call is

```
int status, width, height, beginx, beginy;
int boundx, boundy, boundw, boundh, endx, endy;

status = graf_dragbox(width, height, beginx, beginy, boundx,
   boundy, boundw, boundh, &endx, &endy);
```

where *width* and *height* specify the size of the box. The variables *beginx* and *beginy* contain the starting position of the box, while the variables *boundx, boundy, boundw,* and *boundh* describe the position and size of the boundary rectangle. When the user releases the mouse button, the function returns the ending mouse position in the variables *endx* and *endy.*

The next box function, *graf_slidebox(),* is also used to move a box within a container. This function, however, is object-oriented. The function moves a box object (called a slider) that is located within a parent box object (known as the bar). As with some of the other box functions, the program calls this routine only after it has determined that the user has pressed the left mouse button. Typically, the program would make this call when *form_do()* exits and indicates that the TOUCHEXIT box inside the bar was the exit object. The graf_slidebox( ) moves the slider within the bar whenever the mouse pointer moves toward either end of the parent box. When the user releases the mouse button, the function returns a number indicating the position of the slider within the bar. The calling syntax for graf_slidebox( ) is

```
int status, parent, object, orientation;
OBJECT *tree;
position = graf_slidebox(tree, parent, object, orientation);
```

where *tree* is a pointer to the object tree array containing the two box objects, *parent* is the number of the parent box object, and *object* is the number of the box object which it contains. *Orientation* is a flag which indicates whether the parent bar is oriented horizontally or vertically. If this flag is set to 0, the bar is horizontal and the slider moves left and right. If the flag is set to 1, the bar is vertical and the slider moves up and down. The final position of the slider is returned in the *position* variable. This value is a number in the range 0–1000 which denotes the position of the center of the slider relative to its parent object. If the orientation of the bar is vertical, a position of 0 indicates that the slider is at the top of the bar, while a position of 1000 indicates that it's at the bottom. If its orientation is horizontal, a position of 0 indicates that the slider is at the left of the bar, while a position of 1000 indicates that it's at the right.

Although graf_slidebox( ) draws the dotted outline of a moving box while the user drags the mouse, it does not actually move the slider object after the user lets go. Therefore, your program must move the slider, by calculating a new *object.ob_y* or *object.ob_x* position and by using objc_draw( ) to redraw both the parent slide-bar object and the slider that's contained in it. The calculation is very similar to that used for window sliders. For a vertical slider, the formula would look like this:

```
tree[SLIDER].ob_y = (long)slider_pos *(long)(tree[SLIDEBAR]
   .ob_height - tree[SLIDER].ob_height) /1000;
```

The casts to type *long* are used to avoid overflow in the multiplication and division. Program 7-1 demonstrates how to use the graf_slidebox( ) function in conjunction with a TOUCHEXIT slider object to implement a slide bar in a dialog box. It also demonstrates how to implement user-defined object types. In this dialog box, there are three user-defined selection buttons that are drawn as circles when not selected and as filled circles when selected.

Program 7-1. dialog2.c

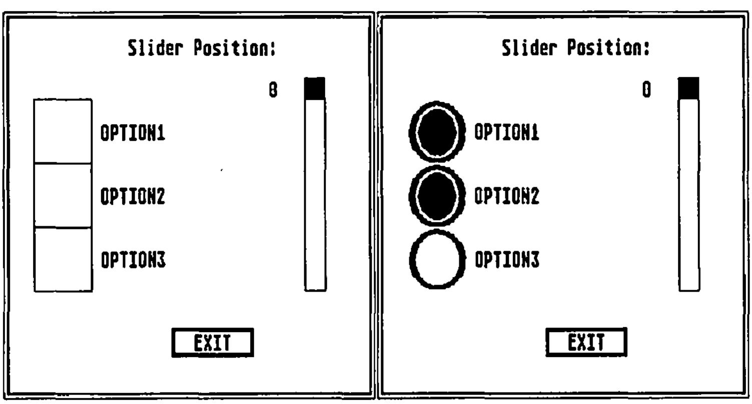
```
/*********************************************************************/
/*                                                                   */
/*      DIALOG2.C -- Demonstrates more sophisticated dialog          */
/*      box, with a slider and user-defined objects.                 */
/*                                                                   */
/*********************************************************************/

#define APP_INFO " "
#define APP_NAME "Dialog Example 2"
#define WDW_CTRLS (NAME)
#define TRUE 1

#include "aesshell.c"
#include "dialog2.h"   /* include file from RSC */

OBJECT *dialtree;   /* pointer to dialog object tree */
TEDINFO *obspec;    /* pointer to TEDINFO for text object */
char *strptr;       /* pointer to text string for text object */
APPLBLK ublock[3];       /* APPLBLK defined in OBDEFS.H  */

demo()
{
   int x, y, width, height, exitbutn, slider_pos;
   int drawcode();

if (!rsrc_load("DIALOG2.RSC"))       /* Load resource file */
   {
   form_alert(3,"[0][Fatal Error!Can't find DIALOS2.RSC file!][Abort]");
   return(0);                     /* Abort if it's not there */
   }

/* get address of menu tree and dialog tree */
   rsrc_gaddr(R_TREE, DIALTREE, &dialtree);

/* get pointer to text objects's text string */
   obspec = (TEDINFO *)dialtree[NUMBER].ob_spec;
   strptr = obspec->te_ptext;  /* pointer to text string */


   for(x=OPTION1;x<(OPTION3+I);x++)
   {
   ublock[x-CPTION1].ub_code = drawcode;        /* set drawcode   */
   dialtree[x].ob_spec = (char *)&ublock[x-OPTION1]; /* to userblock */
   dialtree[x].ob_type = G_PROGDEF;     /* change object type  */
   }


/* Display dialog box */
   form_center(dialtree, &x, &y, &width, &height);
   form_dial(FMD_START, 0, 0, 0, 0, x, y, width, height);
   objc_draw(dialtree, ROOT, MAX_DEPTH, x, y, width, height);

/* Main dialog animation loop */
```

```
/* until exit button hit */
        while((exitbutn = form_do(dialtree, 0)) != EXITBUTN)
                {
                exitbutn &= 0x7fff;  /* mask off top bit of exit object */
                                      /* (set by clicks on TOUCHEXIT) */
  if(exitbutn = SLIDER) /* if exit object was SLIDER */
                       {
/* allow slider to be dragged */
        slider_pos = graf_slidebox(dialtree, SLIDEBAR, SLIDER, 1);
/* set new SLIDER y position (use longs to avoid overflow) */
        dialtree[SLIDER].ob_y = (long)slider_pos *
        (long)(dialtree[SLIDEBAR].ob_height - dialtree[SLIDER].ob_height) /1000
/* redraw slider in new position */
         objc_draw(dialtree, SLIDEBAR, MAX_DEPTH, x, y, width, height);
/* display new postion by changing text of NUMBER object */
         sprintf(strptr,"%4d",slider_pos);   /* redraw text */
/* and redrawing the object */
         objc_draw(dialtree, NUMBER, MAX_DEPTH, x, y, width, height);
                       } /* end of if TOUCHEXIT */

  } /* end of main WHILE loop */

   form_dial(FMD_FINISH, 0, 0, 0, 0, x, y, width, height);


} /* end of DEMO function */

/************ Draw user-defined object *****************************/

drawcode(pb)                        /* Sample user object drawing   */
        PARMBLK *pb;
        {

        int   points[4];
        int   x, y, xr, yr;

        grect_conv(&pb->pb_xc, points);
        vs_clip(handle, TRUE, points);

        vswr_mode(handle,1);
        vsf_color(handle, 1);
        vsf_interior(handle, 0);            /* set fill to hollow */
        if (SELECTED & pb->pb_currstate)    /* If selected */
           vsf_interior(handle, 1);         /* use solid fill pattern */

        vsf_perimeter(handle, 0);
        vsl_width(handle,3);
        xr= pb->pb_w/2;
        yr= pb->pb_h/2;
        x = pb->pb_x + xr;
        y = pb->pb_y + yr;
        if (pb->pb_currstate == pb->pb_currstate)
           v_ellarc(handle, x, y, xr-2, yr-2,0,3599);
        v_ellipse(handle, x, y, xr-8, yr-6);

        return (0);
        }


/* ***************** End of Dialog2.c ************* */
```

In order to run this program, you must first construct the resource file DIALOG2.RSC. If you have a resource construction program, you may do so by following the instructions below. If not, you may use the RSCBUILD program in Appendix C to build the resource from data.

At the top right side is a STRING object whose text reads *Slider Position:*. Directly below that is a text object with the name NUMBER, whose initial text reads *0*. To the right of these objects is the slide bar. It's composed of two box objects. The first, SLIDEBAR, is a long tall G_BOX with an outside border with a thickness of 1 pixel, and it is HOLLOW filled, which means that it is transparent. Inside this object is its child, SLIDER, a G_BOX with an inside border with a thickness of 1. SLIDER is filled with solid black (color 1) and has its TOUCHEXIT flag set. Finally, at the left of the dialog box are three G_BOX items stacked one on top of the other. These are called OPTION1, OPTION2, and OPTION3. They are HOLLOW filled, with an outside border one pixel in thickness, and they have the SELECTABLE flag set. These objects are used as stand-ins for the user-defined objects that will be installed later in the program, since the resource construction programs don't let you create them directly. Each of these box objects has a STRING object to the right of it, whose text reads OPTION*x* where *x* is the option number. Figure 7-1 shows the layout of the dialog as created with the resource construction program (left) and as displayed by the program once the user-defined objects are installed (right).

Figure 7-1. The Dialog Box from Program dialog2.c



As mentioned above, the resource construction programs won't let you build user-defined objects directly. Therefore, you have to use G_OBJECTS and change some fields in the object structure.

First, you've got to change the object type to G_PROGDEF. Next, you must change the *ob_spec* field to a pointer to an APPLBLK structure. This structure should contain the address of your object drawing code in its *ub_code* field. Note also that the object drawing code, *drawcode( ),* only has to draw the outline of the ellipse once. It uses *pb_currstate* and *pb_prevstate* to find out whether the code is being called from *objc_draw( )* or *objc_change( ).* If both fields are the same, objc_draw( ) was called, and the ellipse is drawn. If not, only the filled-ellipse routine is called. The filled-ellipse routine determines whether the circle is blacked in.

Graf_watchbox( ) is the last of the box graphics routines that should be called when the mouse button is first pressed down. This routine simply watches the mouse pointer position while the button is held down and reports whether it ends up inside or outside of a particular object rectangle when the user lets the button up. The graf_watchbox( ) routine is called by form_do( ) to insure that the user, in order to select an object, has both pressed the mouse button while the pointer was over an object and then let the button up while the pointer was still over that object. The most likely use that an applications programmer would make of this routine would be in writing his or her own form_do( ) routine. The syntax for this call is

```
int in_or_out, object, instate, outstate;
OBJECT *tree;
in_or_out = graf_watchbox(tree, object, instate, outstate);
```

where *tree* is a pointer to the object tree and *object* is the number of the objects whose rectangle you wish to check. The *instate* and *outstate* variables should hold the value of the object's ob_state flag when the pointer is inside the object rectangle and when it's outside the rectangle, respectively. For example, a SELECTABLE object will be SELECTED when the pointer is over it and not SELECTED when the pointer is away from it. This allows the routine to change the state of the object and redraw it as the pointer is dragged. When the user lets the left mouse button up, the routine returns a flag in the variable *in_or_out* that indicates whether the pointer ended up inside or outside of the object rectangle. A value of 0 indicates that it was outside the rectangle, while a value of 1 indicates that it ended up inside.

The last box graphics functions are used mainly for cosmetic purposes. The first is used to draw the outline of a box moving from one point onscreen to another. It does this by drawing a series of boxes from the source point to the destination point, one by one, and then erasing those boxes one at a time, in reverse order. All of this happens so quickly that the user gets the impression that something is zooming across the screen. Unless used to indicate a transition from one window another, this effect is mere window dressing, and as such is almost always superfluous. The function used to create the effect is called graf_mbox( ):

```
int status, width, height, beginx, beginy, endx, endy;
status = graf_mbox(width, height, beginx, beginy, endx, endy);
```

where *width* and *height* specify the size of the box, *beginx* and *beginy* its beginning position, and *endx* and *endy* its final position.

Note that the Digital Research documentation refers to this function as *graf_movebox()*. However, the C language bindings released by Atari Corporation with the *Alcyon* C compiler and libraries derived from Atari's code (such as those supplied with the *Megamax* compiler) use the *graf_mbox()* terminology. Therefore, in order to link properly with current versions of the library, your program must also use the graf_mbox( ) form. If, in the future, Atari decides to change its libraries to conform to its documentation, you'll need to change over to graf_movebox( ) as well.

The last two box graphics calls, *graf_growbox()* and *graf_shrinkbox()* are also used mainly for cosmetic purposes. These two calls are very similar to the FMD_GROW and FMD_SHRINK subcommands of the *form_dial()* routine. They draw a set of expanding or shrinking boxes from one screen rectangle to another, like graf_mbox( ) using different size boxes. You can create exploding windows by calling graf_growbox( ) just before *wind_open()*, and graf_shrinkbox( ) just after *wind_close()*. The syntax for these two calls are

```
int status, smallx, smally, smallw, smallh;
int largex, largey, largew, largeh;
status = graf_growbox(smallx, smally, smallw, smallh, largex,
largey, largew, largeh);
status = graf_shrinkbox(largex, largey, largew, largeh, smallx,
smally, smallw, smallh);
```

where *smallx, smally, smallw,* and *smallh* specify the size and position of the smaller rectangle, and *largex, largey, largew,* and *largeh* give the size and position of the larger rectangle. The status variable contains a 0 if an error occurred during the call; otherwise it contains a nonzero integer.

Mouse Form

One of the most commonly used AES Graphics Library call is the one that changes the shape of the mouse pointer. In GEM, the shape of the mouse pointer can be used to indicate what kind of action will take place when the user moves the mouse. For example, a pointing hand is often used for selection or for sizing a rectangle, while a flat hand can indicate that dragging will take place. When a program starts up from the GEM Desktop, the mouse pointer takes the form of the "busy bee," which indicates that the program is busy working, and the user will have to wait until it's finished to begin input.

Normally, the program will change this pointer to the general-purpose arrow shape as soon as the program is ready for input. To do this, it uses the call *graf_mouse( ).* This routine allows the program to set the mouse pointer shape to one of eight predefined forms or to a user-defined 16 × 16–pixel bit-mapped image. It also can be used to temporarily eliminate the mouse pointer display and to restore the pointer again. It is necessary to hide the mouse pointer whenever your program does any drawing that might overwrite it, since the old background behind the pointer will be restored as soon as the pointer is moved. The syntax of the graf_mouse( ) call is

```
int status, form_no, formptr[37];
status = graf_mouse(form_no, formptr);
```

where *form_no* indicates which form the user wishes to install. The valid form numbers and the macro names for them that are found in the GEMDEFS.H file are shown below, along with a short explanation of the typical usage of these forms:

Form Number	Shape	Macro Name	Usage
0	Arrow	ARROW	General purposes
1	Vertical bar (I-beam)	TEXT_CRSR	Text cursor placement
2	Busy bee	HOURGLASS	Busy signal
3	Pointing hand	POINT_HAND	Sizing
4	Flat hand	FLAT_HAND	Dragging
5	Thin crosshairs	THIN_CROSS	Drawing
6	Thick crosshairs	THICK_CROSS	Application-specific
7	Outline crosshairs	OUTLN_CROSS	Application-specific
255	User defined	USER_DEF	Application-specific
256	Mouse pointer off	M_OFF	Hide mouse before drawing
257	Mouse pointer on	M_ON	Restore

When form number 255—the user-defined pointer image—is selected, the *formptr* value should contain the address of a 37-word data array that provides information about the pointer. This information includes the foreground and background colors for the pointer, the shape of the pointer, and a mask which allows you to specify whether the zero bits in the 16 × 16 block are transparent (don't replace existing background with a new color) or opaque (replace existing background with the pointer background color). It also includes the coordinates of the pointer's hot spot or action point. The hot spot is the single pixel which is considered to be the pointer's location on the screen, even though the pointer may be much larger than a single point. On the arrow-shaped pointer, for example, the hot spot is located at the very tip of the arrow. If you click on an icon with the tail of the arrow, rather than with the point, nothing will happen. On the bee-shaped pointer, the hot spot is at the center of the image. The layout of the mouse form definition data structure is as follows:

Element	Meaning
0	*x* position of "hot spot"
1	*y* position of "hot spot"
2	Number of bit planes (must be set to 1)
3	Background color (normally 0)
4	Foreground color (normally 1)
5–20	16 words of color-mask data
21–36	16 words of image data

The x and y coordinates of the hot spot are measured as an offset from the top left corner of the 16 x 16–pixel block. The image data block is laid out so that each line of the image is represented by one 16-bit word, with the most significant bit of the word representing the leftmost dot and the least significant bit representing the rightmost dot. Each bit position that contains a 1 is colored with the foreground pen, and each bit position that contains a 0 is colored either with the background pen, or whatever color is displayed by the existing background, depending on the color mask.

The color mask is used to define the shape of the pointer, without regard to color information. Those bit positions containing a 1 are considered to be inside the pointer. They will be colored with the foreground pen if the corresponding bit positions in the image data also contain a 1. They will be colored with the background pen if the corresponding bit positions in the image data contain a 0. Those bit positions which contain a 0 are considered to be outside the pointer image and, therefore, are transparent. Whatever background image exists on the screen will continue to be displayed at these points.

It is important to use both foreground and background pens in your pointer, in order to insure that it remains visible against any kind of background. Even though the normal system pointers such as the arrow appear to be black only, there is actually a thin white line surrounding them. This makes it possible for the user to see the arrow, even when it's in front of a solid black background.

Program 7-2, written in C, shows how to use evnt_mouse( ) or the mouse rectangle function of evnt_multi( ) to track the mouse pointer and change its shape as it moves into different regions of the screen. It draws a box in the center of the screen and changes the pointer shape as it moves into the areas above, below, to the right, and to the left of the box. When the pointer moves into the box itself, it changes to the user-defined shape whose data is stored in the array called *pointer.* This pointer is supposed to look like the ST mouse itself, though the resemblance is better in high- or low-resolution mode than in medium resolution.

Program 7-2. mousform.c

```
/*****************************************************************/
/*                                                               */
/*      MOUSFORM.C -- shows how to change the mouse pointer      */
/*      shape as the pointer moves around the screen.            */
/*                                                               */
/*                                                               */
/*****************************************************************/

#define APP_INFO " ** Click on the Close Box to exit the program. **"
#define APP_NAME "Mouse Form Demonstration Program"
#define WDW_CTRLS (NAME|CLOSER|INFO)

#include "aesshell.c"

/* Data for our own custom "mouse" pointer */

int pointer[37] =
  {
  0,0,1,    /* x and y of hot spot */
  0,1,      /* background and foreground pens */

/* 16 words of color mask data */
    0xF000, 0xF800, 0x7DC0, 0x3FE0,0x1FF0, 0x1FF8, 0x1FF8, 0x3FFC,
    0x7FFE, 0x3FFF, 0x1FFF, 0x0FFF,0x07FF, 0x01FE, 0x00FC, 0x0078,

/* 16 words of image data */
   0xC000, 0x7000, 0x1080, 0x1D40,0x0E20, 0x0410, 0x0A28, 0x1148,
   0x2084, 0x1102, 0x0A01, 0x0401,0x0302, 0x0084, 0x0048, 0x0030

  };

int points[10]; /* array of points for box */

demo()
{
  int event, done = 0;
  int dummy, msg[8];
  int mx, my;
  GRECT r;              /* mouse rectangle to watch */

/* set points for center rectangle */
   points[0] = points[6] = points[8] = work.g_x + (work.g_w/3);
   points[1] = points[3] = points[9] = work.g_y + (work.g_h/3);
   points[2] = points[4] = work.g_x + (work.g_w * 2 /3);
   points[5] = points[7] = work.g_y + (work.g_h *2/3);

/* draw box, find mouse position, and set pointer shape accordingly */
   graf_mouse(M_OFF, 0L);
   v_pline(handle, 5, points);
   graf_mkstate(&mx, &my, &dummy, &dummy);
   set_ptr(mx, my, &r);
   graf_mouse(M_ON, 0L);

/* Main Program Loop */

   while (! done)   /* until user clicks on close box */
   {                /* check messages and mouse rectangle */
   event = evnt_multi(MU_MESAG|MU_M1,
   0,0,0,           /* evnt_button */
   1, r.g_x, r.g_y, r.g_w, r.g_h,  /* evnt_mouse1 */
   0,0,0,0,0,       /* evnt_mouse2 */
   &msg,            /* evnt_mesg */
   0,0,             /* evnt_timer */
   &mx,&my,         /* mouse x,y */
   &dummy,          /* mouse button */
   &dummy,          /* shift keys */
```

```
    &dummy,          /* evnt_keyboard */
    &dummy);         /* number of clicks */

    if (event & MU_MESAG)  /* if we get a window close message, */
       if (msg[0] == WM_CLOSED)  /* we're done */
           done = 1;

    if (event & MU_M1) /* if mouse leaves rectangle, change pointer */
       set_ptr(mx, my, &r);  /* and reset rectangle to watch */

   } /* end of main WHILE loop */
} /* end of demo() function */

/*************  Pointer change routine ************************/

set_ptr(x, y, r)
   int x, y;  /* mouse x, y */
   GRECT *r;  /* new rectangle to watch */
{
   int form;

   if (y < work.g_y)  /* if above window work area */
      {
      form = 0;        /* set to arrow shape */
      r->g_x = 0;
      r->g_y = 0;
      r->g_w = work.g_w;
      r->g_h = work.g_y-1;
      }

   else if (y < points[1]) /* if above center box */
      {
      form = 3;              /* set to pointing hand */
      r->g_x = work.g_x;
      r->g_y = work.g_y;
      r->g_w = work.g_w;
      r->g_h = work.g_h/3;
      }

   else if(y > points[5])  /* if below center box */
      {
      form = 4;              /* set to open hand */
      r->g_x = work.g_x;
      r->g_y = work.g_y + (work.g_h * 2 / 3);
      r->g_w = work.g_w;
      r->g_h = work.g_h/3;
      }

   else if(x < points[0])  /* if left of center box */
      {
      form = 5;              /* set to thin cross hairs */
      r->g_x = work.g_x;
      r->g_y = work.g_y + (work.g_h / 3);
      r->g_w = work.g_w /3;
      r->g_h = work.g_h/3;
      }

   else if(x > points[2])  /* if right of center box */
      {
      form = 6;              /* set to thick cross hairs */
      r->g_x = work.g_x + (work.g_w *2 / 3);
      r->g_y = work.g_y + (work.g_h / 3);
      r->g_w = work.g_w /3;
      r->g_h = work.g_h/3;
      }
    else                     /* if IN center box */
      {
      form = 255;            /* set pointer to user-defined shape */
```

```
        r->g_x = points[0];   /* (looks like the mouse, sort of) */
        r->g_y = points[1];
        r->g_w = points[4] - points[0] + 1;
        r->g_h = points[5] - points[1] + 1;
        }

    graf_mouse(form, pointer);

} /* end of set_ptr() */

/******************** End of Mousform.c **************/
```

Program 7-3 is a machine language version of the same program.

Program 7-3. mousform.s

```
***********************************************************
*                                                         *
* MOUSFORM.S  -- Shows how to change pointer shape        *
*                                                         *
***********************************************************

*** External references

** Export:

.xdef     demo      * external demo subroutine.
.xdef     wdwctrl
.xdef     wdwtitl
.xdef     wdwinfo

** Import:

.xref     vdi
.xref     aes

.xref     vwkhnd    * virtual workstation handle,
.xref     contrl0   * all of the VDI data arrays
.xref     contrl1
.xref     contrl2
.xref     contrl3
.xref     contrl4
.xref     contrl5
.xref     contrl6
.xref     intin
.xref     ptsin
.xref     ctrl0   * all of the AES data arrays
.xref     ctrl1
.xref     ctrl2
.xref     ctrl3
.xref     ctrl4
.xref     aintin
.xref     aintout
.xref     addrin

.xref     deskx
.xref     desky
.xref     deskw
.xref     deskh
.xref     workx
.xref     worky
.xref     workw
.xref     workh
```

```
  .text

demo:
   move     #0,d4  * close window flag in d4

   jsr      mousoff

   move     #6,contrl0   *opcode for polyline
   move     #5,contrl1   *number of points in ptsin
   move     #0,contrl3   * no integer parameters in intin
   move     vwkhnd,contrl6 *virtual workstation handle

   sub.l    d0,d0
   move     workw,d0     * find left side of box
   divu     #3,d0
   move     d0,thirdw
   add      workx,d0
   move     d0,left
   move     d0,ptsin
   move     d0,ptsin+12
   move     d0,ptsin+16

   add      thirdw,d0    * find right side of box
   move     d0,right
   move     d0,ptsin+4
   move     d0,ptsin+8

   sub.l    d0,d0
   move     workh,d0     * find top side of box
   divu     #3,d0
   move     d0,thirdh
   add      worky,d0
   move     d0,top
   move     d0,ptsin+2
   move     d0,ptsin+6
   move     d0,ptsin+18

   add      thirdh,d0    * find bottom side of box
   move     d0,bottom
   move     d0,ptsin+10
   move     d0,ptsin+14

   jsr      vdi          * draw box

*** find current mouse position
   move     #79,ctrl0    * opcode = graf_mkstate
   move     #0,ctrl1     * no intins
   move     #5,ctrl2     * 5 intout
   move     #0,ctrl3     * 0 addrin
   jsr      aes

*** set pointer shape accordingly
   jsr      setptr

   jsr      mouson

*** main program loop
main:
   move     #25,ctrl0    * opcode = evnt_multi
   move     #16,ctrl1
   move     #7,ctrl2     * 1 intout
   move     #1,ctrl3     * 1 addrin
   move     #0,ctrl4

   move     #20,aintin     * waiting for message or rect. 1
   move.l   #msg,addrin    * message buffer address
   move     #1,aintin+8    * mouse rectangle 1 flag
```

```
   jsr      aes
   move     #0,ctrl3

   move     aintout,d5
   cmpi     #4,d5          * did we get a mouse rectangle event?
   bne      message
   jsr      setptr         * if so, change pointer form
   bra      skip
message:
   move     msg,d6         * if not, check message type
   cmpi     #22,d6         * WM_CLOSED?
   bne      skip
   move     #1,d4          * yes, set close flag
skip:
   cmpi     #0,d4          * check if close flag set
   beq      main           * if not, keep going
   rts

** >>>>>>>>>> End of Main Program Code <<<<<<<<<<<<<<< **


*** turn mouse off or on or change pointer shape

setform:
   move     d7,aintin   * form number
   bra      mousel
mousoff:
   move     #256,aintin * hide the mouse
   bra      mousel
mouson:
   move     #257,aintin * show  the mouse
mousel:
   move     #78,ctrl0   * command = graf_mouse
   move     #1,ctrl1    * 1 input integers
   move     #1,ctrl2    * 1 output integer
   move     #1,ctrl3    * 1 input address
   move.l   #pointer,addrin

   jmp      aes

*** set pointer form and new rectangle to watch

setptr:

   move     aintout+2,d0  * mouse x position
   move     aintout+4,d1  * mouse y position
   move     workw,aintin+14   * next 3 are full window wide
   cmp      worky,d1      * above window work area?
   bhi      setpt1
   move     #0,d7         * if so, change form to arrow
   move     #0,aintin+10
   move     #0,aintin+12
   move     worky,d2
   subq     #1,d2
   move     d2,aintin+16
   bra      setpt6:

setpt1:
   move     thirdh,aintin+16  * all of the rest are 1/3 window high
   move     workx,aintin+10   * next 3 start at left
   cmp      top,d1            * if above center box
   bhi      setpt2:
   move     #3,d7             * set to pointing hand
   move     worky,aintin+12
   bra      setpt6:
```

```
setpt2:
   cmp      bottom,d1          * if below center box
   bcs      setpt3:
   move     #4,d7              * set to open hand
   move     bottom,aintin+12
   bra      setpt6:

setpt3:
   move     thirdw,aintin+14   * next 3 all are 1/3 window wide
   move     top,aintin+12      * and y is at top
   cmp      left,d0            * if at left of center box
   bhi      setpt4:
   move     #5,d7              * set to thin cross hairs
   bra      setpt6:

setpt4:
   cmp      right,d0           * if at right of center box
   bcs      setpt5:
   move     #6,d7              * set to thick cross hairs
   move     right,aintin+10
   bra      setpt6:

setpt5:
   move     #255,d7            * if in center box, set to custom
   move     left,aintin+10     * pointer shape
setpt6:
   jmp      setform

*** Storage space and data constants

.data
.even

msg:     .ds.w 8                 * buffer for message event
left:    .ds.w 1
right:   .ds.w 1
top:     .ds.w 1
bottom:  .ds.w 1
thirdw:  .ds.w 1
thirdh:  .ds.w 1

wdwtitl:  .dc.b 'Mouse Pointer Demonstration',0    * text of window title
wdwinfo:  .dc.b 'Click close box to end ',0    * text of window info line
wdwctrl:  .dc.w 19                 * window control flag

pointer:  .dc.w 0,0,1              * x and y of hot spot
          .dc.w 0,1                * background and foreground pens */
* 16 words of color mask data *
          .dc.w $F000, $F800, $7DC0, $3FE0
          .dc.w $1FF0, $1FF8, $1FF8, $3FFC
          .dc.w $7FFE, $3FFF, $1FFF, $0FFF
          .dc.w $07FF, $01FE, $00FC, $0078

* 16 words of image data *
          .dc.w $C000, $7000, $1080, $1D40
          .dc.w $0E20, $0410, $0A28, $1148
          .dc.w $2084, $1102, $0A01, $0401
          .dc.w $0302, $0084, $0048, $0030

.end
```

Mouse and Keyboard Input

The final graphics routine is used for graphics input—which roughly translates to receiving information from the mouse pointer and shift keys. The *graf_mkstate()* call provides some of the same information as the evnt_button( ) and evnt_multi calls. The difference is that graf_mkstate( ) doesn't wait until an event occurs. Rather, it returns immediately, reporting the current status of the mouse buttons and shift keys. This makes it suitable for use in a polling routine that checks one or both of the mouse buttons. The format used for calling for this routine is

```
int reserved, mousex, mousey, mousbutn, shiftkey;
reserved = graf_mkstate(&mousex, &mousey, &mousbutn,
  &shiftkey);
```

where *mousex* and *mousey* contain the horizontal and vertical coordinates of the mouse pointer, and *mousbutn* and *shiftkey* contain information concerning the mouse button and shift key status. The mouse button state at the time of the call is returned in the mousbutn variable. The value returned is a 1 if the left button is down, 2 if the right button is down, or 3 if both are down. The shiftkey variable contains a code that tells whether the right Shift key, the left Shift key, the Control key, or the Alt key was pressed at the same time as the mouse button. Each of the four low bits represents a different key:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

Thus if shiftkey contains a 4, the Control key was held down when the mouse button was pressed, and if it has a value of 12, both Alt and Control were held down. The *reserved* variable is reserved for future use. Currently, a 1 is always returned in the reserved variable.

Since the current version of ST BASIC doesn't include any commands for checking the mouse, the graf_mkstate( ) function can be very useful to BASIC programmers. The following short sample program shows how to use graf_mkstate from BASIC. It merely waits for the user to press a mouse button and then tells him or her which button was pressed, the location of the mouse, and shift-key status code.

Program 7-4. mous.bas

```
10    apb# = gb
20    control = peek(apb#)
30    global = peek(apb#+4)
40    gintin = peek(apb#+8)
50    gintout = peek(apb#+12)
60    addrin = peek(apb#+16)
70    addrout = peek(apb#+20)
80    fullw 2: clearw 2: gotoxy 1,1
90    print "Press a mouse button to continue":?:?
100   mousbutn = 0
110   while(mousbutn=0)
120   gemsys(79): REM call graf_mkstate until a button is pushed
130   mousex = peek(gintout+2)
140   mousey = peek(gintout+4)
150   mousbutn = peek(gintout+6)
160   shiftkey = peek(gintout+8)
170   wend
180   if mousbutn  > 0 then b$="the left"
190   if mousbutn  > 1 then b$="the right"
200   if mousbutn  > 2 then b$="both"
210   print " You pressed ";b$;" mouse button(s)"
220   print " while the mouse was located at";mousex;",";mousey
230   print " Shift key status was";shiftkey
```

Chapter 8

Desk Accessories

The programs discussed so far have been applications run by double-clicking their icons from the Desktop. As stated early on, however, GEM supports another type of program as well. This type of program is called a *desk accessory,* because of its ability to run concurrently with the Desktop or any other application program that has installed a menu bar.

Desk Accessories

Desk accessory programs are loaded at boot time. These include all programs located in the root directory of the boot disk (either the floppy disk in the A drive or the C partition on the hard disk) which have the extender .ACC attached to their names. There may be a maximum of six of these programs. Desk accessory programs are started by selecting the menu item associated with them from the DESK menu at the far left side of the menu bar.

There are several small but important differences between writing a desk accessory program and writing a normal application.

The first difference is in the initial code used to start up the program. When an application program starts, it is given control over the entire Transient Program Area (TPA), which consists of all of the free system memory. Normally, the application will give back any memory that isn't used for program code, data, or the stack. It does this either in a machine language SETBLOCK call or in a C module that is linked in before the program object module. For example, *Alcyon C* users must always link the file GEMSTART.O or APPSTART.O before the program module. In *Megamax C*, the file INIT.O, which is part of the SYSLIB library, is linked in automatically.

A desk accessory, however, isn't assigned any free memory when it's loaded at boot time. Therefore, a different startup module must be used, so the desk accessory doesn't try to give back extra memory that isn't there. In the case of *Alcyon C*, you must link the module ACCSTART.O first, in place of GEMSTART.O or APPSTART.O. With *Megamax C*, you have to link in the ACC.L library file after your program object file. This will automatically override the INIT.O module in the SYSLIB library.

Since desk accessory programs are started from the DESK menu, they need a way to enter text strings into the menu. They do this using the *menu_register()* call. The syntax for this call is

```
extern int gl_apid;
int menuid;
static char *menutext;

menuid = menu_register(gl_apid, menutext)
```

where *gl_apid* is the application ID number which is assigned to the desk accessory and stored in its global array when it calls *appl_init(). Menutext* is a pointer to the null-terminated character array that holds the text of the menu item.

Normally, the initial lines of a desk accessory program would look like this:

```
appl_init( );
menuid = menu_register(gl_apid, "Accessory Name");
```

The function returns a menu ID number in the *menuid* variable. The ID number is used to identify which menu item was used to start the accessory program. This is necessary because a single accessory can register more than one menu item. For example, a single accessory program displays both the control panel and printer setup menu items. Note, however, that there are only six menu slots available on the DESK menu. When these are used up, no additional desk accessories may be loaded. This means that each additional menu item used by an accessory reduces the total number of desk accessories that may be loaded. For this reason, your accessory should stick to one menu item and use its own system for presenting multiple functions, such as a dialog box.

The AES uses the message system to inform a desk accessory when one of its menu items is selected. Since the desk accessory has no control over the menu bar, it can't use the MN_SELECTED message the way regular applications can. Instead, the AES sends it an AC_OPEN message. The format for this message is as follows:

Word Number	Contents
0	40 (AC_OPEN), the message ID number
4	The menu item number (menuid) of the desk accessory the user selected

The number in word 4 of the message buffer contains the menuid that was returned when the accessory called menu_register( ). You should be aware that this practice differs from Digital Research's GEM documentation, which states that the menu ID is returned in word 3 of the message buffer. (You should also be aware that the GEM documentation shows the message numbers for AC_OPEN and AC_CLOSE to be 30 and 31. Actually, they are 40 and 41, respectively.)

When the user closes the main application program and returns to the Desktop, the desk accessory's windows are automatically closed and deleted. The AES informs the accessory that this has happened with the AC_CLOSE message, which has the following format:

Word Number	Contents
0	41 (AC_CLOSE), the message ID number
3	The menu item number of the desk accessory whose window handles have been lost due to closure of the main application

The main difference between a desk accessory program and an application program, therefore, is that the desk accessory usually doesn't open a window until its menu item is selected and that the desk accessory program never ends and returns to the Desktop. Though the desk accessory may close its windows (or have them closed for it), it always remains loaded in memory, waiting for the user to select its menu item. Therefore, desk accessory programs are structured a bit differently from normal applications. After a short initialization process, during which they call menu_register( ), they enter an endless loop centered around an evnt_multi( ) call, which waits, among other things, for the user to select the accessory's menu item. The following C sample program demonstrates a simple desk accessory. It is an adaptation of the message.c program found in Chapter 3.

Program 8-1. deskacc.c

```
/*********************************************************************/
/*                                                                   */
/*      DESKACC.C -- A modification of MESSAGE.C, to make it        */
/*      work as a desk accessory.                                    */
/*                                                                   */
/*********************************************************************/

#include <osbind.h>     /* Macro definitions for BIOS calls */
#include <gemdefs.h>    /* Flag definitions for Library routines */
#include <obdefs.h>     /* Object definitions */

#define FALSE Ø
#define TRUE 1
#define NONE -1
#define APP_INFO ""
#define APP_NAME "Accessory Window"
#define WDW_CTRLS (NAME|CLOSER|SIZER|MOVER|FULLER)
#define MAX(X,Y) ( (X) > (Y) ? (X) : (Y) )
#define MIN(X,Y) ( (X) < (Y) ? (X) : (Y) )


#define DESK Ø          /* The flag for the Desktop Window */
#define NO_ERR Ø        /* Error no. for "no error" */
#define APP_ERR 1       /* Error no. for failure of appl_init() */
#define VWK_ERR 2       /* Error no. for failure of v_opnvwk() */
#define WDW_ERR 3       /* Error no. for failure of wi_create() */


/* Global variables -- For VDI bindings and program routines */

extern int gl_apid;   /* The application ID part of the global array */
int ap_id;

int contrl[12],         /* VDI data arrays */
    intin[128],
    ptsin[128],
    intout[128],
    ptsout[128];

int phys_handle,    /* workstation handle for physical screen device */
     handle,        /* workstation handle for virtual screen device */
     wi_handle,     /* window handle */
     menuid,        /* menu id number */
     msg[8];        /* message buffer */

int work_in[12],    /* input and output arrays for v_opnvwk() */
    work_out[57];

GRECT desk, work;   /* Desktop and application window dimensions */


int cellw, cellh, chspcw, chspch;  /* size of default character font */


/* Program starts here */

main()
{
    int error;             /* Error flag */

    error = init_acc();  /* Initialize application, open workstation, */
                         /* and open application window */

    if (!error)           /* If no initialization failures,  */
      {
       while (1)                    /* forever... */
          {
          evnt_mesage(&msg);        /* get messages... */
          handle_msg();             /* and handle them. */
```

```
            }  /* end of while */
        } /* end of if */

  } /* end of main */

/*** Initialize accessory, graphics workstation, & register menu ***/

 init_acc()
 {
    int x;

 /* Initialize the GEM application.  If this fails, return error code. */

    appl_init();
    ap_id=gl_apid;
    if (ap_id==-1) return (APP_ERR);

/* Initialize input array, get the physical workstation handle,
   but don't open the Virtual Screen Workstation for VDI calls. */

    phys_handle = graf_handle(&cellw, &cellh, &chspcw, &chspch);
                             /* get physical screen device handle */
    work_in[10]=2;           /* use Raster Coordinates */
    work_in[0]=Getrez()+2;   /* set screen device ID according to */
                             /*  resolution mode.  */
    for (x=1; x<10; work_in[x++]=1);
                             /* set other input values to default */

    menuid = menu_register(ap_id, "  Sample Accessory ");
                             /* register our menu */
    wi_handle = NONE;
    return(0);               /* Report no errors */
 }

/***** Message Handler Routine ****/

handle_msg()                       /* message handler */
{
   long x;

   switch(msg[0])                 /* check message type */
      {

      case AC_OPEN:
        if(msg[4]==menuid)  /* if our menu item was picked */
          {
             if (wi_handle == NONE) /* if there's no window, */
                init_out();         /* create one */
                                    /* else bring it to the front */
             else wind_set(wi_handle, WF_TOP, 0, 0, 0, 0);
          }
          break;

      case AC_CLOSE:
         /* if a window was open, change handle to NONE */
         /* to show that the AES has taken it away from us */
         if( (msg[3]==menuid) && (wi_handle != NONE) )
           {
           v_clsvwk(handle);
           wi_handle = NONE;
           }

      case WM_REDRAW:             /* if redraw, call refresh routine */
         refresh(msg[3], (GRECT *)&msg[4]);
         break;

      case WM_TOPPED:             /* if topped, send to top */
         wind_set(msg[3], WF_TOP, 0, 0, 0, 0);
         break;
```

```
        case WM_SIZED:             /* if sized, check for min size,
                                      then resize */
          msg[6] = MAX(msg[6], cellw*8);
          msg[7] = MAX(msg[7], cellh*4);
          wind_set(msg[3], WF_CURRXYWH, msg[4], msg[5], msg[6], msg[7]);
          redraw_msg(msg[3], (GRECT *)&msg[4]);
          break;

        case WM_MOVED:            /* if moved, make sure the window
                                     stays on the Desktop */
          if (msg[4] + msg[6] > desk.g_x + desk.g_w)
              msg[4] = desk.g_x + desk.g_w - msg[6];

          if (msg[5] + msg[7] > desk.g_y + desk.g_h)
              msg[5] = desk.g_y + desk.g_h - msg[7];

          wind_set(msg[3], WF_CURRXYWH, msg[4], msg[5], msg[6], msg[7]);
          break;

        case WM_FULLED:           /* if fulled, do toggle routine */
          toggle(msg[3]);
          break;

        case WM_CLOSED:           /* if closed, close window and
                                     workstation  */
           if (msg[3] == wi_handle)
              {
              wind_close(msg[3]);
              wind_delete(msg[3]);
              v_clsvwk(handle);
              wi_handle = NONE;
              }
           break;

        default:
          break;
        }
}

/***  Open virtual workstation, create and open output window ***/

init_out()
{
    handle = phys_handle;
    v_opnvwk (work_in, &handle, work_out);
                            /* open virtual screen workstation */
    if (handle == 0)return(VWK_ERR);
                            /* if we can't open it, return error code */

/*  Find out the maximum size for a window, and open one.  */

    wind_get(DESK, WF_WORKXYWH, &desk.g_x, &desk.g_y,
             &desk.g_w, &desk.g_h);
                            /* find dimensions of Desktop Window */
    wi_handle = wind_create(WDW_CTRLS, desk.g_x, desk.g_y,
                             desk.g_w, desk.g_h);
                            /* Create a window that size */
    if (wi_handle<0)
       {
       form_alert(1,"[0][Can't open accessory|No windows left!][OK]");
       v_clsvwk(handle);  /* if we can't, close workstation  */
       return(WDW_ERR);   /* and return error code */
       }

    wind_set(wi_handle,WF_INFO, APP_INFO,0,0);
    wind_set(wi_handle,WF_NAME, APP_NAME,0,0);
                            /* set name and info string for window */
    wind_open(wi_handle, desk.g_x, desk.g_y, desk.g_w/2, desk.g_h/2);
                            /* open the window to quarter size */
```

```
    return(0);              /* report no errors */
}



/****  routine to handle WM_FULLED message ****/

toggle(wh)
   int wh;
{
   GRECT prev, curr, full;

   /* get current, previous, and full size for window */
   wind_get(wh, WF_CURRXYWH, &curr.g_x, &curr.g_y, &curr.g_w, &curr.g_h);
   wind_get(wh, WF_PREVXYWH, &prev.g_x, &prev.g_y, &prev.g_w, &prev.g_h);
   wind_get(wh, WF_FULLXYWH, &full.g_x, &full.g_y, &full.g_w, &full.g_h);

   /* If full, change to previous (unless that was full also) */
   if(((curr.g_x == full.g_x) &&
       (curr.g_y == full.g_y) &&
       (curr.g_w == full.g_w) &&
       (curr.g_h == full.g_h))     &&
      ((prev.g_x != full.g_x) ||
       (prev.g_y != full.g_y) ||
       (prev.g_w != full.g_w) ||
       (prev.g_h != full.g_h)))
     {
      wind_set(wh, WF_CURRXYWH, prev.g_x, prev.g_y, prev.g_w, prev.g_h);
      redraw_msg(wh, &prev); /* send a redraw message, cause AES won't */
      }

   /* If not full, change to full */
   else
      wind_set(wh, WF_CURRXYWH, full.g_x, full.g_y, full.g_w, full.g_h);

}

/***** routine to handle window_refresh (WM_REDRAW) message ******/
refresh(wh, drect)
   int    wh;         /*  window handle from msg[3] */
   GRECT *drect;      /*  pointer to damage rectangle  */
   {
   GRECT   wrect;     /*  the current window rectangle in rect list */

   graf_mouse(M_OFF, 0L);       /* turn off mouse */
   wind_update(BEG_UPDATE);     /* lock screen */

   wind_get                     /* get first rectangle */
    (wh, WF_FIRSTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w, &wrect.g_h);

   while ( wrect.g_w && wrect.g_h ) /* while not at last rectangle,   */
      {
      if (overlap(drect, &wrect))   /* check to see if this one's damaged, */
        {
            set_clip(&wrect);       /* if it is, set clip rectangle */
            display();              /* redraw, and turn clip off */
            vs_clip(handle, FALSE, (int *)&wrect );
        }
      wind_get(wh, WF_NEXTXYWH, &wrect.g_x, &wrect.g_y, &wrect.g_w,
         &wrect.g_h);
      }

   wind_update(END_UPDATE);         /* unlock screen */
   graf_mouse(M_ON, 0x0L);          /* turn mouse pointer back on */
   }

/**** Routine to  draw the window display *****/
display()
{
  int points[4];  /* VDI points array */
```

```
  wind_get(msg[3], WF_WORKXYWH, &work.g_x, &work.g_y,
          &work.g_w, &work.g_h);     /* find work area */
  clear_rect(&work);                 /* and clear it */

  grect_conv(&work, &points);        /* convert work grect to array */
  vsf_interior(handle,2);            /* set fill type to pattern */
  vsf_style(handle, 7 * msg[3] + 2); /* adjust fill pattern */
  vsf_color(handle, msg[3]);         /* set color */
  v_ellipse(handle, points[0] + (work.g_w/2), points[1] + (work.g_h/2),
            work.g_w/2, work.g_h/2); /* draw a filled ellipse */

}
/* >>>>>>>> Utility routines used by other functions <<<<<<<<<<<<< */


set_clip(r)   /* set clip to specified rectangle  */
   GRECT   *r;
   {
   int  points[4];

   grect_conv(r, points);
   vs_clip(handle, TRUE, points);
   }


overlap(r1, r2)       /* compute overlap of two rectangles   */
   GRECT   *r1, *r2;
   {
   int x, y;

   x = MAX(r2->g_x, r1->g_x);
   y = MAX(r2->g_y, r1->g_y);
   r2->g_w = MIN(r2->g_x + r2->g_w, r1->g_x + r1->g_w) -x;
   r2->g_h = MIN(r2->g_y + r2->g_h, r1->g_y + r1->g_h) -y;
   r2->g_x = x;
   r2->g_y = y;
   return( (r2->g_w > 0) && (r2->g_h > 0) );
   }


redraw_msg(wh, r) /* Send Redraw Message to your own window  */
   int    wh;
   GRECT   *r;
   {
   int  msg[8];

   msg[0] = WM_REDRAW;
   msg[1] = gl_apid;
   msg[2] = 0;
   msg[3] = wh;
   msg[4] = r->g_x;
   msg[5] = r->g_y;
   msg[6] = r->g_w;
   msg[7] = r->g_h;
   appl_write(gl_apid, 16, &msg);
   }


/* >>>>>>>>>>>>>>>>>>> Some Handy Functions <<<<<<<<<<<<<<<<< */

clear_rect(r)  /* clear a rectangle to the background color */
   GRECT *r;
{
  int points[4];

  vsf_interior(handle,0);
  grect_conv(r, &points);
  vr_recfl(handle, points);
}
```

```
grect_ccnv(r, array)    /* convert grect to an array of points  */
   GRECT   *r;
   int  *array;
   {
   *array++ = r->g_x;
   *array++ = r->g_y;
   *array++ = r->g_x + r->g_w - 1;
   *array = r->g_y + r->g_h - 1;
   }


/************  End of Deskacc.c **********************/
```

This program combines most of the AESSHELL.C and MESSAGE.C programs into one. The standard AESSHELL.C file could not be included this time because it opens its window right away. In this combined file, only part of the original initialization routine is performed immediately. The virtual window and workstation are opened when the program gets the AC_OPEN message. Also, the cleanup routine is eliminated, since the program never ends. Instead, the evnt_mesage( ) routine is part of an endless loop in which events are received and handled. Most of the event handling is the same, though routines have been added to handle the messages AC_OPEN and AC_CLOSE, and the WM_CLOSED message handler has been changed to close the window and virtual workstation and delete the window.

Remember that linking this program is different from a normal application. With the *Megamax* linker, the program command line reads

MMLINK deskacc.c acc.l -o sample.acc

With *Alcyon* C, the linker command would look like the following:

LINK68 [U] deskacc.68K=accstart,deskacc,vdibind,osbind,aesbind, libf,gemlib,libf

Remember that the executable program must be named with an extender of .ACC instead of .PRG. In this case, the program was given the name SAMPLE.ACC.

A machine language version of this desk accessory program appears below for the benefit of those who choose to program in that language. Since machine language programs contain their own startup code, that code was merely edited, rather than linked with another startup file. The program below contains the startup code and, therefore, should not be linked with the AESSHELL.O object file. Unlike the other machine language examples in this book, it's a complete program by itself.

Program 8-2. deskacc.s

```
**************************************************************
*                                                            *
*  DESKACC.S  -- Assembly language version of desk           *
*  accesory program, drawn from MESSAGE.S                    *
*                                                            *
**************************************************************

*** Program equates

aescode  =  $c8    * command number for AES call
vdicode  =  $73    * command number for VDI call

*** Program starts here.  Set address of our stack.

   .text
      move.l      #accstk,a7     * set stack pointer to our stack

*** Initialize the application with appl_init

   move.l   #0,resv1     * clear global variables
   move.l   #0,resv2
   move.l   #0,resv3
   move.l   #0,resv4
   move     #10,ctrl0    * command = appl_init
   move     #0,ctrl1     * no integer input parameters
   move     #1,ctrl2     * 1 integer output parameter
   move     #0,ctrl3     * no address input parameters
   move     #0,ctrl4     * no address output parameters
   jsr      aes          * do the call

*    cmpi     #$FFFF,apid * check to see if init failed
*    beq      apperr      * and exit if it did

*** Get the physical screen device handle from graf_handle

   move     #77,ctrl0         * command = graf_handle
   move     #5,ctrl2          * 5 integer output parameters
   jsr      aes               * do the call
   move     aintout,pwkhnd    * save handle and char sizes
   move     aintout+2,cellw
   move     aintout+4,cellh
   move     aintout+6,chboxw
   move     aintout+8,chboxh

   move     #$FFFF,wdwhnd     * show that no window is open

*** Register our item on the Desk menu

   move     #35,ctrl0         * command = menu_register
   move     #1,ctrl1          * 1 integer input parameter
   move     #1,ctrl2          * 1 integer output parameter
   move     #1,ctrl3          * 1 address input parameter

   move     apid,aintin       * application id
   move.l   #menutxt,addrin   * address of menu text item
   jsr      aes
   move     aintout,menuid
```

```
*** Main accessory loop
***
main:
   move      #23,ctrl0     * opcode = evnt_mesage
   move      #0,ctrl1
   move      #1,ctrl2      * 1 intout
   move      #1,ctrl3      * 1 addrin
   move      #0,ctrl4
   move.l    #msg,addrin
   jsr       aes
   move      #0,ctrl3

   jsr       msghand       * handle the message
   bra       main          * and loop back

** >>>>>>>>>> End of Main Program Code <<<<<<<<<<<<<<< **


*** Initialize workstation and window
***
initout:

*** Open the Virtual Screen Workstation (v_opnvwk)
   move      #100,contrl0   * opcode to contrl(0)
   move      #0,contrl1     * no points in ptsin
   move      #11,contrl3    * 11 integers in intin
   move      pwkhnd,contrl6 * physical workstation handle to contrl(6)

   movea.l   #intin+2,a0    * destination address
   move      #8,d0          * loop counter
initloop:
   move.w    #1,(a0)+       * intin(1)-intin(9) = 1
   dbra      d0,initloop

   move      #2,intin+20    * intin(10) = 2 (Raster Coordinates)
   move.w    #4,-(sp)       * push getrez command on stack
   trap      #14            * call XBIOS
   addq.l    #2,sp          * pop command off stack
   addq      #2,d0
   move      d0,intin       * use rez+2 as device ID

   jsr       vdi
   move      contrl6,vwkhnd * save virtual workstation handle
*   beq       vwkerr          * end program if it's zero

*** Find max window size

   move      #104,ctrl0     * command = wind_get
   move      #2,ctrl1       * 2 input integers
   move      #5,ctrl2       * 5 output integers
   move      #0,aintin      * window handle of Desktop
   move      #4,aintin+2    * WF_WORKXYWH command

   jsr       aes
   move      aintout+2,deskx  * store desk x,y,w,h
   move      aintout+4,desky
   move      aintout+6,deskw
   move      aintout+8,deskh

***  Create a window with max size

   move      #100,ctrl0   * command = wind_create
   move      #5,ctrl1     * 5 input integers
   move      #1,ctrl2     * 1 ouput integer
   move      wdwctrl,aintin * window ctrl flag
   move      deskx,aintin+2 * max x
   move      desky,aintin+4 * max y
   move      deskw,aintin+6 * max width
   move      deskh,aintin+8 * max height
```

```
    jsr     aes
    move    aintout,wdwhnd * save window handle
*   bmi     wdwerr          * if negative, exit program

*** set window name

    move    #105,ctrl0  * command = wind_set
    move    #6,ctrl1    * 6 input integers
    move    wdwhnd,aintin  * window handle
    move    #2,aintin+2    * subcommand = set window name
    move.l  #wdwtitl,aintin+4 * point to title

    jsr     aes

*** Open the window to 1/4 max size

    move    #101,ctrl0  * command = wind_open
    move    #5,ctrl1
    move    #1,ctrl2    * 1 ouput integers
    move    deskx,aintin+2 * initial x
    move    desky,aintin+4 * initial y
    move    deskw,d0
    asr     #1,d0
    move    d0,aintin+6 * initial width
    move    deskh,d0
    asr     #1,d0
    move    d0,aintin+8 * initial width

    jsr     aes

*** Find window work area size

    move    #104,ctrl0  * command = wind_get
    move    #2,ctrl1    * 2 input integers
    move    #5,ctrl2    * 5 ouput integers
    move    #4,aintin+2    * WF_WORKXYWH command

    jsr     aes
    move    aintout+2,workx  * store work x,y,w,h
    move    aintout+4,worky
    move    aintout+6,workw
    move    aintout+8,workh
    rts


*** Message handler subroutine ****
***
msghand:
    move    msg,d5        * check message type
    cmpi    #41,d5        * AC_CLOSE?
    bgt     msg5          * if greater, exit
    bne     msg00         * if less, try next
    move    msg+6,d0      * is msg(3) = menuid?
    cmp     menuid,d0
    bne     msg5          * if not, exit
    cmpi    #$FFFF,wdwhnd  * is window handle -1?
    beq     msg5          * if no window, exit
    move    #$FFFF,wdwhnd  * signal that there's no window
    jmp     clsvwk        * close virtual workstation

msg00:
    cmpi    #40,d5        * AC_OPEN?
    bne     msg0          * if not, try next message type
    move    msg+8,d0      * is this our menuid?
    cmp     menuid,d0
    bne     msg5            * if not, exit
    cmpi    #$FFFF,wdwhnd  * is no window open?
    bne     msg30           * if not, move window to top
    jmp     initout         * if so, open vwk and window
```

```
msgØ:
    cmpi      #28,d5         * WM_MOVED?
    beq       msg1:          * if so, change size and position
    cmpi      #27,d5         * WM_SIZED?
    bne       msg2

msg1:
    move      #1Ø5,ctrlØ        * command = wind_set
    move      #6,ctrl1          * 6 input integers
    move      msg+6,aintin      * window handle
    move      #5,aintin+2       * subcommand = set current size
    move      msg+8,aintin+4
    move      msg+1Ø,aintin+6
    move      msg+12,aintin+8
    move      msg+14,aintin+1Ø
    jmp       aes

msg2:
    cmpi      #22,d5         * WM_CLOSED?
    bne       msg3

*** Close the Window
    move      #1Ø2,ctrlØ   * command = wind_close
    move      #1,ctrl1
    move      #1,ctrl2
    move      #Ø,ctrl3
    move      #Ø,ctrl4
    move      wdwhnd,aintin

    jsr       aes

*** Delete the Window

    move      #1Ø3,ctrlØ   * command = wind_delete

    jsr       aes
    move      #$FFFF,wdwhnd  * turn on "no window" flag
    jmp       clsvwk

msg3:
    cmpi      #21,d5         * WM_TOPPED?
    bne       msg4

msg3Ø:
    move      #1Ø5,ctrlØ        * command = wind_set
    move      #6,ctrl1          * 6 input integers
    move      msg+6,aintin      * window handle
    move      #1Ø,aintin+2      * subcommand = WF_TOP

    jmp       aes
msg4:
    cmpi      #2Ø,d5         * WM_REDRAW?
    bne       msg5
    jsr       refresh

msg5:
    rts

*** Window refresh subroutine ***
refresh:
* turn mouse off

    move      #78,ctrlØ    * command = graf_mouse
    move      #1,ctrl1     * 1 input integers
    move      #1,ctrl2     * 1 output integer
    move      #256,aintin  * hide the mouse

    jsr       aes

* lock screen
```

```
    move    #107,ctrl0  * command = wind_update
    move    #1,ctrl1    * 1 input integers
    move    #1,ctrl2    * 1 output integer
    move    #1,aintin   * code = BES_UPDATE

    jsr     aes

* Find first window rectangle

    move    #104,ctrl0  * command = wind_get
    move    #2,ctrl1    * 2 input integers
    move    #5,ctrl2    * 5 ouput integers
    move    msg+6,aintin * window handle
    move    #11,aintin+2 * WF_FIRSTXYWH command

    jsr     aes

refresh1:                       * check for empty rectangle
    move    aintout+6,d0
    or      aintout+8,d0
    beq     refresh3            * if empty, at end, so quit

    move    msg+8,d0
    move    aintout+2,d1
    cmp     d0,d1               * x = MAX (x1, x2)
    bcs     olap1
    move    d1,d0               * overlap x is in d0

olap1:
    move    msg+10,d1
    move    aintout+4,d2
    cmp     d1,d2
    bcs     olap2
    move    d2,d1               * overlap y is in d1

olap2:
    move    msg+8,d2
    add     msg+12,d2
    move    aintout+2,d3
    add     aintout+6,d3
    cmp     d2,d3
    bhi     olap3
    move    d3,d2           * d2 = MIN(x1+w1, x2+w2)

olap3:
    sub     d0,d2
    move    d2,aintout+6    * overlap w = d2 - overlap x

    move    msg+10,d2       * d2 = y1
    add     msg+14,d2       * + h1
    move    aintout+4,d3    * d3 = y2
    add     aintout+8,d3    * + h2
    cmp     d2,d3
    bhi     olap4
    move    d3,d2           * d2 = MIN(y1+h1, y2+h2)

olap4:
    sub     d1,d2
    move    d2,aintout+8    * overlap h = d2 - overlap y

    or      aintout+6,d2    * are width and height both 0?
    beq     refresh2        * if so, skip redraw and get next rect

* set clip rectangle

    move    #129,contrl0    * opcode for set clip (vs_clip)
    move    #2,contrl1      * two points in ptsin
    move    #0,contrl2
    move    #1,contrl3
    move    #0,contrl4
    move    #1,intin        * turn clipping on
```

```
   move    d0,ptsin            * points[0] = overlap x
   add     aintout+6,d0
   subq    #1,d0               * points[2] = overlap x+w -1
   move    d0,ptsin+4
   move    d1,ptsin+2          * points[1] = overlap y
   add     aintout+8,d1
   subq    #1,d1
   move    d1,ptsin+6          * points[3] = overlap y+h -1

   jsr     vdi

* redraw the display
   jsr display

* turn clipping off

   move    #129,contrl0        * opcode for set clip (vs_clip)
   move    #2,contrl1          * two points in ptsin
   move    #0,contrl2
   move    #1,contrl3
   move    #0,contrl4
   move    #0,intin            * turn clipping off

   jsr     vdi

* get next window rectangle

refresh2:
   move       #104,ctrl0  * command = wind_get
   move       #2,ctrl1    * 2 input integers
   move       #5,ctrl2    * 5 ouput integers
   move       msg+6,aintin * window handle
   move       #12,aintin+2 * WF_NEXTXYWH command

   jsr        aes
   bra        refresh1

* unlock screen

refresh3:
   move       #107,ctrl0  * command = wind_update
   move       #1,ctrl1    * 1 input integers
   move       #1,ctrl2    * 1 output integer
   move       #0,aintin   * code = END_UPDATE

   jsr        aes

* turn mouse on

   move       #78,ctrl0   * command = graf_mouse
   move       #1,ctrl1    * 1 input integers
   move       #1,ctrl2    * 1 output integer
   move       #257,aintin * hide the mouse

   jmp        aes


*** Window display subroutine ***

display:
* Find window work area size

   move       #104,ctrl0  * command = wind_get
   move       #2,ctrl1    * 2 input integers
   move       #5,ctrl2    * 5 ouput integers
   move       msg+6,aintin
   move       #4,aintin+2    * WF_WORKXYWH command
```

```
    jsr     aes
    move    aintout+2,workx  * store work x,y,w,h
    move    aintout+4,worky
    move    aintout+6,workw
    move    aintout+8,workh

* set fill pattern to hollow

    move    #23,contrl0    * opcode for set fill type
    move    #0,contrl1
    move    #0,contrl2
    move    #1,contrl3     * one integer in intin
    move    #1,contrl4
    move    #0,intin       * select hollow fill type

    jsr     vdi

* clear work area of window

    move    #114,contrl0    * opcode for fill rectangle (vr_recfl)
    move    #2,contrl1      * two points in ptsin
    move    #0,contrl3
    move    #0,contrl4

    move    workx,d0
    move    d0,ptsin
    add     workw,d0
    subq    #1,d0
    move    d0,ptsin+4
    move    worky,d0
    move    d0,ptsin+2
    add     workh,d0
    subq    #1,d0
    move    d0,ptsin+6

    jsr     vdi

* set fill type to pattern

    move    #23,contrl0    * opcode for set fill type
    move    #0,contrl1
    move    #1,contrl4
    move    #2,intin       * select pattern fill type

    jsr       vdi

* set type of fill pattern

    move    #24,contrl0    * opcode for set fill style (vsf_style)
    move    msg+6,d0       * window number...
    mulu    #7,d0          * x 7
    add     #2,d0          * + 2
    move    d0,intin       * pattern type

    jsr       vdi

* set fill color

    move    #25,contrl0    * opcode for set fill color (vsf_color)
    move    msg+6,d0       * window number...
    addq    #1,d0          * + 1
    move    d0,intin       * is color number

    jsr       vdi

* draw an ellipse
```

```
   move    #11,contrl0   * opcode for GDP
   move    #5,contrl5    * sub-code for ellipse
   move    #2,contrl1    * two points in ptsin
   move    #0,contrl3
   move    #0,contrl4

   move    workw,d0      * take window width
   asr     #1,d0         * divide in half
   move    d0,ptsin+4    * for horiz. radius of circle
   add     workx,d0      * add left x of window
   move    d0,ptsin      * for center of circle
   move    workh,d0      * take window height
   asr     #1,d0         * divide in half
   move    d0,ptsin+6    * for vert. radius of circle
   add     worky,d0      * add top y of window
   move    d0,ptsin+2    * for center of circle

   jmp     vdi


*** Close Virtual Screen Workstation subroutine (v_clsvwk)
***

clsvwk:
   move      #101,contrl0   * opcode to contrl(0)
   move      #0,contrl1     * no points in ptsin
   move      #0,contrl3     * no integers in intin

   jmp       vdi

***   Make AES function call
***   (after setting parameters)

aes:
   move.l   #apb,d1
   move.w   #aescode,d0
   trap    #2
   rts

*** Make VDI function call
*** (after setting parameters)

vdi:
   move.l   #vpb,d1
   move.w   #vdicode,d0
   trap    #2
   rts

*** Storage space for program stack, AES and VDI call parameters,
*** and miscellaneous program variables

      .bss
      .even
                 .ds.l       256     * program stack
accstk:           .ds.l       1

********** VDI Data Arrays ************

contrl:
contrl0:     .ds.w 1
contrl1      .ds.w 1
contrl2:     .ds.w 1
contrl3:     .ds.w 1
contrl4:     .ds.w 1
contrl5:     .ds.w 1
contrl6:     .ds.w 1
contrl7:     .ds.w 1
contrl8:     .ds.w 1
contrl9:     .ds.w 1
contrl10:    .ds.w 1
contrl11:    .ds.w 1
```

```
intin:      .ds.w 128
intout:     .ds.w 128
ptsin:      .ds.w 128
ptsout:     .ds.w 128


************** AES Data Arrays ****************
ctrl:
ctrl0:    .ds.w 1
ctrl1     .ds.w 1
ctrl2:    .ds.w 1
ctrl3:    .ds.w 1
ctrl4:    .ds.w 1

global:
version:    .ds.w 1
count:      .ds.w 1
apid:       .ds.w 1
private:    .ds.l 1
tree:       .ds.l 1
resvl:      .ds.l 1
resv2:      .ds.l 1
resv3:      .ds.l 1
resv4:      .ds.l 1

aintout:    .ds.w 8
aintin:     .ds.w 18
addrin:     .ds.l 3
addrout:    .ds.l 2


****** Misc variables *********

vwkhnd    .ds.w 1
pwkhnd    .ds.w 1
wdwhnd    .ds.w 1

chboxw    .ds.w 1
chboxh    .ds.w 1
cellw     .ds.w 1
cellh     .ds.w 1

deskx     .ds.w 1
desky     .ds.w 1
deskw     .ds.w 1
deskh     .ds.w 1

workx     .ds.w 1
worky     .ds.w 1
workw     .ds.w 1
workh     .ds.w 1

msg:      .ds.w 8                   * buffer for message event
menuid    .ds.w 1

************** Initialized data constants

.data
.even
menutxt:  .dc.b '  Sample Accessory ',0  * text of menu item
wdwtitl:  .dc.b 'Accessory Window',0     * text of 1st window title
wdwctrl:  .dc.w 43                  * window control flag

*** The AES and VDI parameter blocks hold pointers
*** to the starting address of each of the data arrays

apb:  .dc.l ctrl,global,aintin,aintout,addrin,addrout
vpb:  .dc.l contrl,intin,ptsin,intout,ptsout


.end
```

You should be aware of certain problems with desk accessories in the current (preblitter) version of GEM. First, *evnt_timer()* behaves somewhat unpredictably from a desk accessory and may cause the program to lock up. The same is true of the MU_TIMER portion of *evnt_multi()*. Also, the normal *form_do()* routine has a tendency to let keystrokes "fall through" from the accessory to the application. This means that if your accessory uses a dialog box with editable text fields, some of the keystrokes may not reach those fields, but instead, they may end up being sent to the main application. You should consider writing your own form_do( ) in order to get accurate text entry from a desk accessory dialog box, at least with the current version of GEM on the ST.

Appendix A

AES Function Reference

Initialize Application

appl_init( ) Opcode=10

This call registers the application with the AES, which then initializes several elements in the application's global data array. One of these contains the ID number which the AES assigns to the application. This ID number is used by other tasks, such as the GEM Screen Manager, when they wish to communicate with the application through its message buffer.

C binding

```
int ap_id;
    ap_id = appl_init( );
```

Note: As of this writing, the C bindings do not return the correct value in ap_id. Since this value is returned in the global variable gl_apid, the following work-around may be used:

```
extern int gl_apid;
int ap_id;
    appl_int( );
    ap_id = gl_apid;
```

Inputs

	control[0] = 10	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array

Results

ap_id	int_out[0] =	Application ID number

See also

appl_exit( )

Read Message Pipe

appl_read( ) Opcode=11

Reads a specified number of bytes from the application's message pipe.

C binding

```
int status, ap_id, length, msgbuf[ ];
    status = appl_read(ap_id, length, msgbuf);
```

Inputs

	control[0] = 11	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
ap_id	int_in[0] =	ID of application whose message pipe is to be read (normally the application's own)
length	int_in[1] =	Number of bytes to read from message pipe
msgbuf	addr_in[0] =	Address of the buffer used to store the bytes that are read from the message pipe

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

appl_write( ), appl_init( )

Write to Message Pipe

appl_write( ) Opcode=12

Writes a specified number of bytes to an application's message pipe.

C binding

```
int status, ap_id, length, msgbuf[ ];
    status = appl_write(ap_id, length, msgbuf);
```

Inputs

	control[0] = 12	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
ap_id	int_in[0] =	ID of application whose message pipe was written to (normally another application's ID)
length	int_in[1] =	Number of bytes to written to message pipe
msgbuf	addr_in[0] =	Address of the buffer where the bytes to write are stored

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

appl_read( ), appl_init( )

Find Application ID

appl_find( ) Opcode=13

Finds the application ID number of a named application that is currently running in the system. The ID number may be used to establish communications with this application, via the message pipe.

C binding

```
int id;
char name[8];
   id = appl_find(name);
```

Inputs

	control[0] = 13	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
name	addr_in[0] =	Address of text string containing the name of the application to be found. This text string must contain exactly eight text characters, followed by the NULL (ASCII 0) character. If the name of the application contains fewer than eight characters, spaces must be added to the end of the name to pad it to eight characters.

Results

ap_id	int_out[0] =	The application ID number of the application that was named. If the application is not currently loaded, −1 is returned.

Note: It has been reported that sometimes this function will return a valid ID number even after the application requested has been closed.

See also

appl_write( ), appl_init( )

Playback Mouse and Key Macro

appl_tplay( ) **Opcode=14**

Plays a recording that has been made of the user's mouse and keyboard input. This function does not work under the current version of GEM, but should be fixed in future versions.

C binding

```
int actions, speed;
char buffer[ ];

    appl_tplay(buffer, actions, speed);
```

Inputs

	control[0] = 14	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
actions	int_in[0] =	The number of user actions to play back
speed	int_in[1] =	The speed at which to play them back, on a scale of 1–10,000, where 100 equals the original speed at which the actions were performed, 50 equals half speed, 200 equals double speed, and so on.
buffer	addr_in[0] =	Address of the buffer used to store the recorded user actions

Results

int_out[0] =	Always equals 1

See also

appl_trecord( )

Record Mouse and Key Macro

appl_trecord( ) **Opcode=15**

Records a specified number of the user's mouse and keyboard input actions. This function does not work under the current version of GEM, but should be fixed in future versions.

C binding

```
int actions;
char buffer[ ];

    appl_trecord(buffer, actions);
```

Inputs

	control[0] = 15	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
actions	int_in[0] =	The number of user actions to be recorded. Since each action requires six bytes of storage, this number should be no greater than the size of the buffer divided by 6.
buffer	addr_in[0] =	The address of the buffer where the users recorded actions will be stored. Each event is stored as six bytes. The first two bytes hold a 16-bit event code: 0 = timer event 1 = mouse button event 2 = mouse movement event 3 = keyboard event The last four bytes are a 32-bit longword the meaning of which depends on the event: Timer: Time elapsed (in milliseconds) Mouse Button: Low word = button state High word = number of clicks Mouse Movement: Low word = *x* position High word = *y* position Keyboard: Low word = character typed High word = shift-key status

Results

recorded	int_out[0] =	The number of events actually recorded

See also

appl_tplay( )

Clean Up Application

appl_exit( ) Opcode=19

This function notifies the AES that the application is about to terminate, so that the AES can release whatever system resources are allocated to the application.

C binding

```
int status;
    status = appl_exit( );
```

Inputs

	control[0] = 19	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

appl_init( )

Wait for Keyboard Event

evnt_keybd( ) Opcode=20

Waits for the user to press any key and returns the appropriate keycode.

C binding

```
int keycode;
   keycode = event_keybd( );
```

Inputs

	control[0] = 20	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array

Results

keycode	int_out[0] =	Keycode value for the key pressed (See Appendix B for meaning of keycodes.)

See also

evnt_multi( )

Wait for Mouse Button Event

evnt_button( ) **Opcode=21**

Waits for the user to press a particular combination of mouse buttons a specified number of times and returns information about the mouse-button state and the shift-key state.

C binding

int clicked, clicks, bmask, bstate;
int mousex, mousey, button, shiftkey;

clicked = evnt_button(clicks, bmask, bstate, &mousex, &mousey, &button, &shiftkey);

Inputs

	control[0] = 21	Opcode
	control[1] = 3	Number of 16-bit inputs in int_in array
	control[2] = 5	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
clicks	int_in[0] =	The number of clicks to wait for. To be more precise, the number of times that the mouse-button state must match the bstate flag within a set time period before the function returns. The time period is set by the evnt_dclick( ) call.
bmask	int_in[1] =	The mouse buttons for which the operation is waiting: 1 = left mouse button 2 = right mouse button 3 = both mouse buttons
bstate	int_in[2] =	The button state for which the application is waiting: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down

Results

clicked	int_out[0] =	The number of times the mouse-button state actually matched bstate. This is always a number between 1 and the value stored in the variable clicks.
mousex	int_out[1] =	The horizontal position of the mouse pointer at the end of the function
mousey	int_out[2] =	The vertical position of the mouse pointer at the end of the function

button	int_out[3] =	The final mouse-button state: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down
shiftkey	int_out[4] =	The status of the keyboard shift keys. Each key is represented by a different bit. A 1 in that bit position means that the key is down, while a 0 means that it's up:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

See also

evnt_multi( )

Wait for Mouse Rectangle Event

evnt_mouse( ) **Opcode=22**

This function waits for the mouse pointer to leave or enter a particular rectangle on the display screen.

C binding

```
int reserved, mflag, rectx, recty, rectw, recth,
    mousex, mousey, button, shiftkey;

    reserved = evnt_mouse (mflag, rectx, recty, rectw, recth, &mousex,
&mousey, &button, &shiftkey);
```

Inputs

	control[0] = 22	Opcode
	control[1] = 5	Number of 16-bit inputs in int_in array
	control[2] = 5	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
mflag	int_in[0] =	A code which specifies whether the call is waiting for the mouse pointer to enter the rectangle, or leave the rectangle: 0 = waiting for mouse to enter rectangle 1 = waiting for mouse to leave rectangle
rectx	int_in[1] =	The left coordinate of the rectangle, in screen pixels
recty	int_in[2] =	The top coordinate of the rectangle, in screen pixels
rectw	int_in[3] =	The width of the rectangle, in screen pixels
recth	int_in[4] =	The height of the rectangle, in screen pixels

Results

reserved	int_out[0] =	Reserved for future use; always equals 1
mousex	int_out[1] =	The final horizontal position of the mouse pointer
mousey	int_out[2] =	The final horizontal position of the mouse pointer
button	int_out[3] =	The final mouse button state: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down
shiftkey	int_out[4] =	The status of the keyboard shift keys. Each key is represented by a different bit. A 1 in that bit position means that the key is down, while a 0 means that it's up:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

See also

evnt_multi( )

Wait for Message Event

evnt_mesag( ) Opcode=23

This function waits for a standard 16-byte message to appear in the message pipe. When the message appears, it reads the 16 bytes into a buffer.

Messages can be used by one task to communicate to another task, or even to itself. For example, the AES Screen Manager task sends standard messages to let an application know when one of its menu items has been selected or when one of its windows needs to be redrawn.

C binding

```
int reserved, msgbuf[8];
    reserved = evnt_mesag(msgbuf);
```

Inputs

	control[0] = 23	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
msgbuf	addr_in[0] =	The address of a 16-byte buffer in which the AES will store the message

Results

reserved	int_out[0] =	Reserved for future use; always equals 1

Upon return from this function, a 16-byte message will be stored in the buffer pointed to by msgbuf. The general format for AES messages is

Element Number	Contents
0	Message ID (indicates type of message)
1	Application ID of message sender
2	Number of additional bytes in message (in excess of the standard 16)
3–7	Message-dependent

There are a number of standard AES messages. The specific formats for these message are

Word Number	Contents
0	10 (MN_SELECTED message). A menu item was selected by the user.
3	The object number of the menu title that was selected.
4	The object number of the menu item that was selected.

Word Number	Contents
0	20 (WM_REDRAW message). A window display needs to be redrawn
3	The handle of the window whose display needs refreshing
4	The *x* position of the damage rectangle
5	The *y* position of the damage rectangle
6	The width of the damage rectangle
7	The height of the damage rectangle

Word Number	Contents
0	21 (WM_TOPPED message). The user selected a new window to be active.
3	The handle of the window the user selected to be active

Word Number	Contents
0	22 (WM_CLOSED message). The user clicked on the Close Box.
3	The handle of the window whose close box was clicked

Word Number	Contents
0	23 (WM_FULLED message). The user clicked on the Full Box.
3	The handle of the window whose full box was clicked

Word Number	Contents
0	24 (WM_ARROWED message). The user clicked on a scroll bar or arrow.
3	The handle of the window whose scroll bar or arrow was clicked
4	The action requested by the user: 0 = page up (user clicked on scroll bar above vertical slider) 1 = page down (user clicked on scroll bar below vertical slider) 2 = line up (user clicked on up arrow) 3 = line down (user clicked on down arrow) 4 = page left (user clicked on scroll bar left of horizontal slider) 5 = page right (user clicked on scroll bar right of horizontal slider) 6 = column left (user clicked on left arrow) 7 = column right (user clicked on right arrow)

Word Number	Contents
0	25 (WM_HSLID message). The user wants to move the horizontal slider.
3	The handle of the window whose horizontal slider was dragged
4	The requested position for the left edge of the slider (a number in the range 0–1000, where 0 = far left, 1000 = far right)

Word Number	Contents
0	26 (WM_VSLID message). The user want to move the vertical slider.
3	The handle of the window whose vertical slider was dragged
4	The requested position for the top edge of the slider (a number in the range 0–1000, where 0 = top, 1000 = bottom)

Word Number	Contents
0	27 (WM_SIZED message). The user has dragged the Size Box.
3	The handle of the window for which the size change is requested
4	The requested *x* position of the window's left edge (the same as the current window *x* position)
5	The requested *y* position of the window's top edge (the same as the current window *y* position)
6	The requested width of the window
7	The requested window height

Word Number	Contents
0	28 (WM_MOVED message). The user has dragged the Move Bar.
3	The handle of the window whose move bar was dragged
4	The requested *x* position of the window's left edge
5	The requested *y* position of the window's top edge
6	The requested width of the window (the same as the current width)
7	The requested window height (the same as the current window height)

Word Number	Contents
0	29 (WM_NEWTOP message). A window has become active.
3	The handle of the window that's become active

Word Number	Contents
0	40 (AC_OPEN message). A desk accessory menu has been selected.
4	The menu item number (menuid) of the desk accessory the user selected

Word Number	Contents
0	41 (AC_CLOSE message). An application has closed, so desk accessories should release their window handles.
3	The menu item number of the desk accessory to be closed

See also

evnt_multi( )

Wait for Timer Event

evnt_timer( ) Opcode=24

This functions waits for a specified number of milliseconds to pass before returning.

C binding

```
int reserved;
unsigned int timelo, timehi,
    reserved = evnt_timer(timelo, timehi);
```

Inputs

	control[0] = 24	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
timelo	int_in[0] =	Low word of the 32-bit timer value
timehi	int_in[1] =	High word of the 32-bit timer value Together, the low and high timer words form a 32-bit timer value which specifies the period of time the function should wait, in milliseconds.

Results

reserved	int_out[0] =	Reserved for future use; always returns a 1

See also

evnt_multi( )

Wait for Multiple Event

evnt_multi( ) Opcode=25

This function allows the application to wait for multiple event types at the same time. The application can specify that it wishes to wait for keyboard events, mouse-button events, up to two mouse-rectangle events, message events, and/or timer events. The function returns as soon as any one of the specified events occurs. When the call ends, the mouse *x* and *y* position, mouse-button state, and shift-key state are returned, regardless of the type of events requested.

C binding

```
int happened, events, clicks, bmask, bstate;
int m1flag, m1rectx, m1recty, m1rectw, m1recth;
int m2flag, m2rectx, m2recty, m2rectw, m2recth;
int msgbuf[8];
int timelo, timehi;
int mousex, mousey, button, shiftkey;
int keycode, clicked;

happened = evnt_multi(events, clicks, bmask, bstate, m1flag, m1rectx,
   m1recty, m1rectw, m1recth, m2flag, m2rectx, m2recty, m2rectw, m2recth,
   msgbuf, timelo, timehi, &mousex, &mousey, &button, &shiftkey,
   &keycode, &clicked);
```

Inputs

	control[0] = 25	Opcode
	control[1] = 16	Number of 16-bit inputs in int_in array
	control[2] = 7	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
events	int_in[0] =	A code which specifies the type of event for which the application is waiting. Each event type is represented by a single bit. If that bit is set to 1, the function will include that event type in the list of events for which it is waiting. The bit values for the various events are

Bit	Bit Value	Macro Name	Event
0	1	MU_KEYBD	Keyboard
1	2	MU_BUTTON	Mouse button
2	4	MU_M1	Mouse rectangle #1
3	8	MU_M2	Mouse rectangle #2
4	16	MU_MESAG	Message
5	32	MU_TIMER	Timer

clicks	int_in[1] =	The number of clicks to wait for. To be more precise, the number of times that the mouse-button state must match the state flag within a set time period before the

		function returns. The time period is set by the evnt_dclick( ) call.
bmask	int_in[2] =	The mouse buttons for which the application is waiting: 1 = left mouse button 2 = right mouse button 3 = both mouse buttons
bstate	int_in[3] =	The button state for which the application is waiting: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down
m1flag	int_in[4] =	A code which specifies whether the call is waiting for the mouse pointer to enter or leave the first mouse rectangle: 0 = waiting for mouse to enter rectangle 1 = waiting for mouse to leave rectangle
m1rectx	int_in[5] =	The left coordinate of the first mouse rectangle, in screen pixels
m1recty	int_in[6] =	The top coordinate of the first mouse rectangle, in screen pixels
m1rectw	int_in[7] =	The width of the first mouse rectangle, in screen pixels
m1recth	int_in[8] =	The height of the first mouse rectangle, in screen pixels
m2flag	int_in[9] =	A code which specifies whether the call is waiting for the mouse pointer to enter or leave the second mouse rectangle: 0 = waiting for mouse to enter rectangle 1 = waiting for mouse to leave rectangle
m2rectx	int_in[10] =	The left coordinate of the second mouse rectangle, in screen pixels
m2recty	int_in[11] =	The top coordinate of the second mouse rectangle, in screen pixels
m2rectw	int_in[12] =	The width of the second mouse rectangle, in screen pixels
m2recth	int_in[13] =	The height of the rectangle, second mouse in screen pixels
timelo	int_in[14] =	Low word of the 32-bit timer value
timehi	int_in[15] =	High word of the 32-bit timer value Together, the low and high timer words form a 32-bit timer value which specifies the period of time the function should wait, in milliseconds.
msgbuf	addr_in[0] =	The address of a 16-byte buffer in which the AES will store the message

Results

happened	int_out[0] =	A code which specifies the type of event which actually happened. The code used is identical to that used for events above.
mousex	int_out[1] =	The final horizontal position of the mouse pointer
mousey	int_out[2] =	The final vertical position of the mouse pointer
button	int_out[3] =	The final mouse-button state: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down
shiftkey	int_out[4] =	The status of the keyboard shift keys. Each key is represented by a different bit. A 1 in that bit position means that the key is down, while a 0 means that it's up:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

keycode	int_out[5] =	The code number for the key combination pressed by the user. See Appendix B for a complete list of keycodes.
clicked	int_out[6] =	The number of times the mouse-button state actually matched bstate. This is always a number between 1 and the value stored in the variable clicks.

See also

evnt_keybd( ), evnt_button( ), evnt_mouse( ), evnt_mesag( ), evnt_timer( )

Set Double-Click Speed

evnt_dclick( ) Opcode=26

Reads the current setting of the double-click interval (the amount of time which a mouse button event will wait for multiple clicks) or changes that setting.

C binding

int speed_set, speed, flag;
    speed_set = evnt_dclick(speed, flag);

Inputs

	control[0] = 26	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
speed	int_in[0] =	The new double-click speed. A number in the range 0–4, where 0 is the slowest speed and 4 the fastest speed.
flag	int_in[1] =	A code which specifies whether you wish to read the current setting or to change it: 0 = set a new speed 1 = read the current speed

Results

speed_set	int_out[0] =	The existing or new double-click setting

See also

evnt_mouse( ), evnt_multi( )

Display or Erase Menu Bar

menu_bar( ) Opcode=30

This function is used to display an application's menu bar or to erase that menu bar. The menu erase function should always be used before changing menu bars and before the appl_exit( ) call made prior to exiting the program.

C binding

```
int status, showflag;
OBJECT *tree;
   status = menu_bar(tree, showflag);
```

Inputs

	control[0] = 30	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
showflag	int_in[0] =	A code which determines whether this call causes the menu bar to be displayed or erased: 0 = erase the menu bar 1 = display the menu bar
tree	addr_in[0] =	The address of the object tree array which supplies the data for the menu display

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Display or Erase Checkmark

menu_icheck( ) Opcode=31

Displays a checkmark in front of a menu item or erases the checkmark.

C binding

```
int status, item, setting;
OBJECT *tree;
   status = menu_icheck(tree, item, setting);
```

Inputs

	control[0] = 31	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
item	int_in[0] =	The object number of the menu item
setting	int_in[1] =	A code indicating whether the specified item should be checked or have its checkmark erased: 0 = erase a checkmark, if visible 1 = display a checkmark in front of the item
tree	addr_in[0] =	The address of the object tree array which supplies the data for the menu display

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

appl_exit( )

Enable or Disable Menu Item

menu_ienable( ) **Opcode=32**

Disables the selection of a menu item or enables its selection. When the menu item is disabled, its text is "grayed out," that is, printed in faint characters.

C binding

```
int status, item, setting;
OBJECT *tree;
   status = menu_ienable(tree, item, setting);
```

Inputs

	control[0] = 32	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
item	int_in[0] =	The object number of the item to be disabled or enabled
setting	int_in[1] =	A code which indicates whether selection of the menu item is to be disabled or enabled: 0 = selection of the menu item is disabled 1 = selection of the menu item is enabled
tree	addr_in[0] =	The address of the object tree array which supplies the data for the menu display

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Toggle Menu Title Highlight

menu_tnormal( ) **Opcode=33**

This function either displays a menu title in normal video or highlights it in reverse video. It is most often used to return a menu title to normal video after the program has carried out an action in response to the selection of a menu item.

C binding

```
int status, title, setting;
OBJECT *tree;
   status = menu_tnormal(tree, title, setting);
```

Inputs

	control[0] = 33	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
title	int_in[0] =	The object number of the menu title which is to be displayed normally or highlighted
setting	int_in[1] =	A code which indicates whether the menu title should be highlighted in reverse video, or displayed in normal video: 0 = display in reverse video 1 = display in normal video
tree	addr_in[0] =	The address of the object tree array which supplies the data for the menu display

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Change Menu Item Text

menu_text( ) Opcode=34

Changes the text of a menu item. This allows menu selections to change in response to changes in the program context.

C binding

```
int status, item;
char *text;
OBJECT *tree;
   status = menu_text(tree, item, text);
```

Inputs

	control[0] = 34	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 2	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
item	int_in[0] =	The object number of the menu items whose text will be replaced
tree	addr_in[0] =	The address of the object tree array which supplies the data for the menu display
text	addr_in[1] =	The address of the new text string to be used for the menu item. The text string should end in the NULL character (ASCII 0) and should be no longer than the string which it replaces.

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Add Item to Desk Menu

menu_register( ) Opcode=35

This function allows a desk accessory to add an item to the DESK menu. A single accessory may add more than one item to the menu, but all accessories may only use a total of six menu items.

C binding

```
extern int gl_apid;
int menuid;
static char *menutext;

    menuid = menu_register(gl_apid, menutext)
```

Inputs

	control[0] = 35	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
gl_apid	int_in[0] =	The application ID number. This value should be returned by appl_init( ), but currently the bindings only place it in the global variable gl_apid.
menutext	addr_in[0] =	The starting address of the null-terminated text string used for the menu item

Results

menuid	int_out[0] =	The desk accessory's menu ID number. Valid ID numbers in the range 0–5. An ID number of −1 means that there is no room on the menu for this item.

Add an Object to a Tree

objc_add( ) **Opcode=40**

Adds an object to an object tree, by linking it in with the parent object and other sibling objects, if any. This function is not normally used by the applications programmer if a resource construction program is used to create the object trees.

binding

```
int status, parent, child;
struct object tree[ ];
   status = objc_add(tree, parent, child);
```

nputs

	control[0] = 40	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
parent	int_in[0] =	The object number of the parent to which a child object will be added
child	int_in[1] =	The object number of the child object to be added
tree	addr_in[0] =	The address of the object tree array which contains the objects
Results		
status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

objc_delete( )

Delete an Object from a Tree

objc_delete( ) Opcode=41

Deletes a child object from an object tree by unlinking it from its parent object and sibling objects, if any. This function is not normally used by the applications programmer if a resource construction program is used to create the object trees.

C binding

```
int status, object;
struct object tree[ ];

    status = objc_delete(tree, object);
```

Inputs

	control[0] = 41	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The object number of the object to be deleted
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

objc_add( )

Draw an Object Tree

objc_draw( ) Opcode=42

This function draws all objects in an entire object tree or in any branch of the tree. It also allows the specification of a clipping rectangle and will only draw those objects which fall within the rectangle.

C binding

```
int status, firstob, depth, clipx, clipy, clipw, cliph;
struct object tree[ ];

    status = objc_draw(tree, firstob, depth, clipx, clipy, clipw, cliph);
```

Inputs

	control[0] = 42	Opcode
	control[1] = 6	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
firstob	int_in[0] =	The object number of the first object in the tree to be drawn. All descendants of this object will be drawn as well.
depth	int_in[1] =	The number of levels of descent from the original object to draw. The object itself is zero levels down, its immediate children are one level down, their children are two levels down, and so on.
clipx	int_in[2] =	The horizontal coordinate for the left side of the clipping rectangle
clipy	int_in[3] =	The vertical coordinate for the top edge of the clipping rectangle
clipw	int_in[4] =	The width of the clipping rectangle, in screen pixels
cliph	int_in[5] =	The height of the clipping rectangle, in screen pixels
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Find Objects Under Mouse Pointer

objc_find( ) Opcode=43

Finds the topmost object that covers a particular point on the screen, usually the spot occupied by the mouse pointer.

C binding

```
int foundob, firstob, depth, x, y;
struct object tree[ ];

    foundob = objc_find(tree, firstob, depth, x, y);
```

Inputs

	control[0] = 43	Opcode
	control[1] = 4	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
firstob	int_in[0] =	The object number of the first object in the tree to be searched. All descendants of this object will be searched as well.
depth	int_in[1] =	The number of levels of descent from the original object to search. The object itself is zero levels down, its immediate children are one level down, their children are two levels down, and so on.
x	int_in[2] =	The horizontal screen coordinate of the point to be searched
y	int_in[3] =	The vertical screen coordinate of the point to be searched
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

foundob	int_out[0] =	The object number of the object found at the designated screen location. If no object was found there, −1 is returned

See also

wind_find( )

Find Object's Screen Position

objc_offset( ) **Opcode=44**

This function calculates the absolute screen position of an object. This function is needed because each object's position is stored internally as an offset relative to that of its parent object.

C binding

```
int status, object, x, y;
struct object tree[ ];

    status = objc_offset(tree, object, &x, &y);
```

Inputs

	control[0] = 44	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 3	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The object whose screen location you wish to find
tree	addr_in[0] =	The address of the object tree array which contains the object

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
x	int_out[1] =	The horizontal screen position of the object, in pixels
y	int_out[2] =	The vertical screen position of the object in pixels

Reorder Child Objects

objc_order( ) Opcode=45

Moves a child object to a new position, relative to its siblings.

C binding

int status, object, newpos;
struct object tree[ ];

status = objc_order(tree, object, newpos);

Inputs

	control[0] = 45	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The object number of the child to be moved
newpos	int_in[1] =	A code specifying the new position at which to place the object: −1 = on top 0 = on the bottom 1 = one from the bottom 2 = two from the bottom (and so on)
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

objc_add( ), objc_delete( )

Let User Edit Text Objects

objc_edit( ) Opcode=46

This function allows the user to edit the text that appears in a G_FTEXT or G_FBOXTEXT object. It is used by the AES as part of the form_do( ) call and is not ordinarily a function that would be used unless the application programmer was writing his or her own version of form_do( ).

C binding

```
int status, object, char, index, type;
struct object tree[ ];
    status = objc_edit(tree, object, char, &index, type);
```

Note: This binding varies from the format specified by the Digital Research documentation, which adds a parameter for the ending index to the end of the parameter list. The binding shown above, however, is the one actually supplied by Atari with the *Alcyon* C compiler and all compilers whose bindings derive from Atari's.

Inputs

	control[0] = 46	Opcode
	control[1] = 4	Number of 16-bit inputs in int_in array
	control[2] = 2	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The object number of the editable text object
char	int_in[1] =	The text character entered by the user
index	int_in[2] =	The position of the next character to be entered in the text string (cursor position)
type	int_in[3] =	A code specifying the type of operation to perform. Valid code numbers include the following:

Type Number	Macro Name	Description of Function
0	ED_START	Reserved for future use
1	ED_INIT	Combine the template string of TEDINFO field te_ptmplt with the text string of the te_ptext field to display the formatted string and then turn the cursor on.
2	ED_CHAR	Check the input character against the validation string in TEDINFO field te_pvalid, update the te_ptext field if the input character is valid, and display the changed text.
3	ED_END	Turn off the text cursor.

tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
index	int_out[0] =	The new position of the next character to be entered in the text string (cursor position) after objc_edit( ) has been performed

See also

form_do( )

Change Object's State Flag

objc_change( ) Opcode=47

This function is used to change an object's ob_state flag. Since this change may affect the object's appearance on screen, the function allows the programmer to request a redraw of the object when its state is changed. A clipping rectangle may also be specified, and only objects within this rectangle are redrawn.

C binding

int status, object, reserved, clipx, clipy, clipw, cliph, state, redraw;
struct object tree[ ];

status = objc_change(tree, object, reserved, clipx, clipy, clipw, cliph, state, redraw);

Inputs

	control[0] = 47	Opcode
	control[1] = 8	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The object number of the object whose state flag is to be changed
reserved	int_in[1] =	Reserved for future use; must always be 0
clipx	int_in[2] =	The horizontal coordinate for the left side of the clipping rectangle
clipy	int_in[3] =	The vertical coordinate for the top edge of the clipping rectangle
clipw	int_in[4] =	The width of the clipping rectangle, in screen pixels
cliph	int_in[5] =	The height of the clipping rectangle, in screen pixels
state	int_in[6] =	The new value for the object's ob_state flag
redraw	int_in[7] =	A code which specifies whether or not the object should be redrawn after the state change: 0 = no redraw of the object 1 = the object is redrawn
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Handle Dialog

form_do( ) Opcode=50

Form_do( ) is like a small subprogram that monitors the user's interaction with a dialog box. It handles the selection of objects with the left mouse button and also handles the entry of text into the editable text objects. Since form_do( ) itself calls evnt_multi( ), it takes control of all event waiting. This means that menus do not function while form_do( ) is executing.

C binding

```
int exitobj, editobj;
OBJECT *tree;
exitobj = form_do(tree, editobj);
```

Inputs

	control[0] = 50	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
editobj	int_in[0] =	The object number of the editable text field at which the cursor should be positioned when the dialog is initially displayed. If there aren't any editable text fields in the dialog, the number 0 should be used.
tree	addr_in[0] =	The address of the object tree array which is used to draw the dialog

Results

exitobj	int_out[0] =	The object number of the object whose selection caused the dialog to terminate. If a TOUCHEXIT object was double-clicked, the high bit of this value will be set.

See also

form_button( ), form_keybd( ), objc_edit( ), evnt_multi( )

Begin or End Dialog

form_dial( ) **Opcode=51**

This function can be used to reserve a portion of the screen for the dialog box, to release that portion of the screen, and to draw an expanding box before the dialog opens, and a contracting one after it finishes.

C binding

int status, type, smallx, smally, smallw, smallh;
int largex, largey, largew, largeh;
status = form_dial(type, smallx, smally, smallw, smallh, largex, largey, largew, largeh);

Note: This binding varies from the one described in the Digital Research documentation, which omits the rectangle information for the second rectangle. The binding shown above, however, conforms to the format actually used in *Alcyon* C bindings supplied by Atari and all bindings that derive from them (such as those used by *Megamax* C).

Inputs

	control[0] = 51	Opcode
	control[1] = 9	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
type	int_in[0] =	A code which specifies the type of action to take:

Type Number	Macro Name	Action
0	FMD_START	Reserves the screen area used by the dialog box
1	FMD_GROW	Draws expanding box from small to large rectangle
2	FMD_SHRINK	Draws shrinking box from large to small rectangle
3	FMD_FINISH	Frees the screen area used by the dialog box and causes redraw messages to be sent

smallx	int_in[1] =	The horizontal coordinate of the left edge of the smaller rectangle
smally	int_in[2] =	The vertical coordinate of the top edge of the smaller rectangle
smallw	int_in[3] =	The width of the smaller rectangle, in screen pixels
smallw	int_in[4] =	The height of the smaller rectangle, in screen pixels
largex	int_in[5] =	The horizontal coordinate of the left edge of the larger rectangle
largey	int_in[6] =	The vertical coordinate of the top edge of the larger rectangle

largew	int_in[7] =	The width of the larger rectangle, in screen pixels
largeh	int_in[8] =	The height of the larger rectangle, in screen pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

form_do( )

Display an Alert Box

form_alert( ) **Opcode=52**

Displays an alert box and returns the user's response to the alert.

C binding

```
int exitbutn, default;
char *string;
   exitbutn = form_alert(default, string);
```

Inputs

	control[0] = 52	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
default	int_in[0] =	The default exit button, which will be selected if the user presses the Return key: 0 = no default exit button 1 = first exit button 2 = second exit button 3 = third exit button
string	addr_in[0] =	The address of the null-terminated string containing the alert text. The format for this string is

[Icon_number][Message text][Exit button text]

This string is separated into three parts, each set off by square brackets. The first item, icon_number, is a single numeric digit indicating which image (if any) should be displayed at the left side of the alert box. The choices are

Icon Number	Image	Meaning
0	None	
1	Exclamation point in diamond	Note
2	Question mark in yield-sign triangle	Wait
3	Octagonal stop sign	Stop

The second set of square brackets holds the text message. This message is limited to a maximum of five lines, each of which may contain a maximum of 40 characters. The vertical bar character (|) is used to indicate the start of a new line.

The final set of square brackets contains the text for the exit buttons. A maximum of three exit buttons may be used, each of which contains a maximum of 20 characters of text. The text for each button is separated with a vertical bar character.

Results

exitbutn	int_out[0] =	The exit button selected by the user: 1 = first exit button 2 = second exit button 3 = third exit button

Display an Error Box

form_error( ) Opcode=53

Displays an error box, which informs the user of a TOS error.

C binding

```
int exitbutn, error;
   exitbutn = form_error(error);
```

Inputs

	control[0] = 53	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
error	int_in[0] =	The TOS error code. Actually, since GEM was designed with IBM PC-DOS in mind, this routine expects to get PC-DOS error codes, rather than the GEMDOS error codes returned on the Atari ST computers. In order for form_error( ) to print out its error messages correctly, you must convert the GEMDOS codes to PC codes. This is done by reversing the sign of the code from negative to positive, and then subtracting 31 (DOS_ERR = (−TOS_ERR) − 31). The following chart lists the GEMDOS error codes for which form_error( ) prints error-specific messages messages (as opposed to *TOS error #X*) and gives the complete text of those messages.

GEMDOS Error Number	PC-DOS Error Number	Error	form_error( ) Message
−33	2	File not found	This application can't find the folder or file you just tried to access.
−34	3	Path not found	This application can't find the folder or file you just tried to access.
−35	4	File handle pool exhausted (no file handles left)	This application doesn't have room to open another document. To make room, close any document that you don't need.

GEMDOS Error Number	PC-DOS Error Number	Error	form_error( ) Message
−36	5	Access denied (wrong attribute or access code)	An item with this name already exists in the directory, or this item is set to Read-only status.
−39	8	Insufficient memory	There isn't enough memory in your computer for the application you just tried to run.
−41	10	Invalid environment	There isn't enough memory in your computer for the application you just tried to run.
−42	11	Invalid format	There isn't enough memory in your computer for the application you just tried to run.
−46	15	Invalid drive specification	The drive you specified does not exist. Check the drive's identifier or change the drive identifier in the DISK INFORMATION dialog.
−47	16	Attempted to remove the current directory	You cannot delete the folder in which you are working.
−49	18	No more files	This application can't find the folder or file you just tried to access.

Results

exitbutn	int_out[0] =	The exit button that the user selected. Since the current version of GEM on the ST only displays one button, this number isn't significant.

Center the Dialog Box

form_center( ) **Opcode=54**

Changes the *x* and *y* coordinates of the root object in a dialog tree, so that the dialog box is centered onscreen. The function also returns the position and size information for the centered dialog.

C binding

int x, y, width, height;
OBJECT *tree;
reserved = form_center(tree, &x, &y, &width, &height);

Inputs

	control[0] = 54	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 5	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
tree	addr_in[0] =	The address of the object tree array which makes up the dialog box

Results

reserved	int_out[0] =	Reserved for future use; always equals 1
x	int_out[1] =	The horizontal coordinate of the left edge of the centered dialog rectangle
y	int_out[2] =	The vertical coordinate of the top edge of the centered dialog rectangle
width	int_out[3] =	The width of the centered dialog rectangle, in screen pixels
height	int_out[4] =	The height of the centered dialog rectangle, in screen pixels

Handle form_do( ) Events

form_keybd( ) Opcode=55

This is the routine form_do( ) calls after its evnt_multi( ) call detects a keypress. If this routine detects a control key like the Tab or Up Arrow, it select the next editable text object. If it detects the Return key, it selects the DEFAULT object. Otherwise, it filters the keystroke and passes the printing key back to the calling routine. Since it is such a limited subset of form_do( ) functions, it is of interest mainly to programmers writing their own form_do( ) routine.

C binding

There is no official C binding for this routine. Therefore, if you wish to call it from C, you must either write a machine language function to call it, write your own bindings, or place the correct values into the global arrays directly and then call the crys_if(55) function which is in the regular bindings. If you choose either of the latter two, you must also change the values in entry 55 of the ctrl_cnts array (which is in the regular bindings, in the file GEMSTART.S for the *Alcyon* compiler) to

.dc.b 3,3,1 * func 55

Inputs

	control[0] = 55	Opcode
	control[1] = 3	Number of 16-bit inputs in int_in array
	control[2] = 3	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
edit_obj	int_in[0] =	Number of the text object currently being edited
next_obj	int_in[1] =	Flag for change in edited object number
keyin	int_in[2] =	Keycode received from evnt_multi
tree	addr_in[0] =	The address of the object tree array which makes up the dialog box

Results

	int_out[0] =	continue
	int_out[1] =	obj_out
	int_out[2] =	keyout
continue	int_out[0] =	Flag for exit object selection 0 = exit object selected 1 = exit object not selected, continue
obj_out	int_out[1] =	Number of new edit object
keyout	int_out[2] =	Processed keycode. A 0 indicates that this function has handled it, and no further processing is required for this key.

See also

form_do( ), form_butn( )

Handle form_do( ) Mouse Events

form_butn( ) **Opcode=56**

This function is called by form_do( ) when its evnt_multi( ) call detects a mouse-button press. First, objc_find( ) is used to locate the object. Then form_butn( ) is called. This routine highlights the object if it's SELECTABLE and not DISABLED and performs the deselect function for radio buttons. It sets an exit flag for EXIT or TOUCHEXIT objects. If the object was EDITABLE, it returns the initial object number and, if not, it zeros it out so that form_do( ) won't change the object. Since this is such a limited subset of form_do( ) functions, it's of interest mainly to programmers writing their own form_do( ) routine.

C binding

There is no official C binding for this routine. Therefore, if you wish to call it from C, you must either write a machine language function to call it, write your own bindings, or place the correct values into the global arrays directly and then call the crys_if(56) function, which can be found in the regular bindings. If you choose either of the latter two, you must also change the values in entry 56 of the ctrl_cnts array (which is in the regular bindings, in the file GEMSTART.S for the *Alcyon* compiler) to

```
.dc.b    2,2,1 * func 56
```

Inputs

	control[0] = 56	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 2	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	Number of the object clicked on
clicks	int_in[1] =	Number of mouse-button clicks received from evnt_multi( )
tree	addr_in[0] =	The address of the object tree array which makes up the dialog box

Results

continue	int_out[0] =	Flag for exit object selection 0 = exit object selected 1 = exit object not selected, continue
obj_out	int_out[1] =	Number of the new edit object

See also

form_do, form_keybd( )

Draw a Rubber Box

graf_rubberbox( ) Opcode=70

This function draws a dotted box outline on the screen, the upper left corner of which is fixed, but the lower right portion of which follows the mouse pointer so long as the user holds down the left mouse button. When the left mouse button is released, the function ends, and the box is erased. This function should only be called when the program has determined that the left mouse button is already down, by returns from form_do( ), evnt_multi( ), or evnt_button( ), since if it is up, the function will end as soon as it's called.

C binding

```
int status, x, y, minw, minh, endw, endh;
    status = graf_rubberbox(x, y, minw, minh, &endw, &endh);
```

Inputs

	control[0] = 70	Opcode
	control[1] = 4	Number of 16-bit inputs in int_in array
	control[2] = 3	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
x	int_in[0] =	The horizontal screen coordinate of the left edge of the box
y	int_in[1] =	The vertical screen coordinate of the top edge of the box
minw	int_in[2] =	The box's minimum width, in pixels
minh	int_in[3] =	The box's minimum height, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
endw	int_out[1] =	The width of the box at the time the user released the left mouse button
endh	int_out[2] =	The height of the box at the time the user released the left mouse button

See also

form_do( ), evnt_multi( ), evnt_button( )

Let the User Drag a Box

graf_dragbox( ) **Opcode=71**

This function draws a dotted box outline on the screen, which stays a fixed distance from the mouse pointer so long as the user holds down the left mouse button. This box is dragged within a boundary rectangle defined by the program. When the left mouse button is released, the function ends, and the box is erased. This function should only be called when the program has determined that the left mouse button is already down, by returns from form_do( ), evnt_multi( ), or evnt_button( ), since, if it's up, the function will end as soon as it's called.

C binding

int status, width, height, beginx, beginy;
int boundx, boundy, boundw, boundh, endx, endy;

status = graf_dragbox(width, height, beginx, beginy, boundx, boundy, boundw, boundh, &endx, &endy);

Inputs

	control[0] = 71	Opcode
	control[1] = 8	Number of 16-bit inputs in int_in array
	control[2] = 3	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
width	int_in[0] =	The width of the box, in pixels
height	int_in[1] =	The height of the box, in pixels
beginx	int_in[2] =	The horizontal position of the left edge of the box at the beginning of the call
beginy	int_in[3] =	The vertical position of the top edge of the box at the beginning of the call
boundx	int_in[4] =	The horizontal position of the left edge of the boundary rectangle. The box outline cannot be dragged past the borders of this imaginary screen rectangle.
boundy	int_in[5] =	The vertical position of the top edge of the boundary rectangle
boundw	int_in[6] =	The width of the boundary rectangle, in pixels
boundh	int_in[7] =	The height of the boundary rectangle, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution; >0 = no error occurred during execution

endx	int_out[1] =	The screen position of the box's left edge at the time the user released the left mouse button
endy	int_out[2] =	The screen position of the box's top edge at the time the user released the left mouse button

See also

form_do( ), evnt_multi( ), evnt_button( )

Draw a Moving Box

graf_mbox( ) **Opcode=72**

Draws and erases a series of box outlines to give the impression of a box moving from one position onscreen to another.

C binding

```
int status, width, height, beginx, beginy, endx, endy;
    status = graf_mbox(width, height, beginx, beginy, endx, endy);
```

Note: Digital Research documentation refers to this function as graf_movebox( ). However, the C language bindings released by Atari Corporation with the *Alcyon* C compiler and libraries derived from Atari's code (such as those supplied with the *Megamax* C compiler) use the graf_mbox( ) terminology. Therefore, in order to link properly with current versions of the library, your program must also use the graf_mbox( ) form. If in the future Atari decides to change its libraries to conform to its documentation, you'll need to change over to graf_movebox( ) as well.

Inputs

	control[0] = 72	Opcode
	control[1] = 6	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
width	int_in[0] =	The width of the box, in pixels
height	int_in[1] =	The height of the box, in pixels
beginx	int_in[2] =	The screen position of the left edge of the first rectangle
beginy	int_in[3] =	The screen position of the top edge of the first rectangle
endx	int_in[4] =	The screen position of the left edge of the final rectangle
endy	int_in[5] =	The screen position of the top edge of the final rectangle

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Draw an Expanding Box

graf_growbox( ) Opcode=73

Draws and erases a series of increasingly larger boxes, to give the appearance of a box expanding from a small size to a larger size. May be used, for example, before opening a window with the wind_open( ) call, to make the window look as if it's exploding open.

C binding

int status, smallx, smally, smallw, smallh;
int largex, largey, largew, largeh;

status = graf_growbox(smallx, smally, smallw, smallh, largex, largey, largew, largeh);

Inputs

	control[0] = 73	Opcode
	control[1] = 8	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
smallx	int_in[0] =	The screen position of the left edge of the starting (small) rectangle
smally	int_in[1] =	The screen position of the top edge of the starting (small) rectangle
smallw	int_in[2] =	The width of the starting (small) box, in pixels
smallh	int_in[3] =	The height of the starting (small) box, in pixels
largex	int_in[4] =	The screen position of the left edge of the ending (large) rectangle
largey	int_in[5] =	The screen position of the top edge of the ending (large) rectangle
largew	int_in[6] =	The width of the ending (large) box, in pixels
largeh	int_in[7] =	The height of the ending (large) box, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_open( ), graf_shrinkbox( )

Draw a Contracting Box

graf_shrinkbox( ) Opcode=74

Draws and erases a series of increasingly smaller boxes, to give the appearance of a box contracting from a large size to a smaller size. May be used, for example, before closing a window with wind_close( ), to make it look like the window is actually folding in on itself.

C binding

int status, smallx, smally, smallw, smallh;
int largex, largey, largew, largeh;

status = graf_shrinkbox(largex, largey, largew, largeh, smallx, smally, smallw, smallh);

Inputs

	control[0] = 74	Opcode
	control[1] = 8	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
smallx	int_in[0] =	The screen position of the left edge of the ending (small) rectangle
smally	int_in[1] =	The screen position of the top edge of the ending (small) rectangle
smallw	int_in[2] =	The width of the ending (small) box, in pixels
smallh	int_in[3] =	The height of the ending (small) box, in pixels
largex	int_in[4] =	The screen position of the left edge of the starting (large) rectangle
largey	int_in[5] =	The screen position of the top edge of the starting (large) rectangle
largew	int_in[6] =	The width of the starting (large) box, in pixels
largeh	int_in[7] =	The height of the starting (large) box, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_close( ), graf_growbox( )

Watch an Object Rectangle

graf_watchbox( ) Opcode=75

This function follows the mouse pointer as it moves in and out of a specified object rectangle, so long as the user holds down the left mouse button. The caller may specify that the object changes state when the mouse pointer moves on or off of it, and the function will redraw the object each time the state change occurs. When the left mouse button is released, the function ends, and it returns a code indicating whether the mouse pointer ended up within the object rectangle or outside. It should only be called when the program has determined that the left mouse button is already down, by returns from form_do( ), evnt_multi( ), or evnt_button( ). If it's up, the function will end as soon as it's called. This function is used internally by form_do( ) to handle object selection.

C binding

```
int in_or_out, object, instate, outstate;
OBJECT *tree;

    in_or_out = graf_watchbox(tree, object, instate, outstate);
```

Inputs

	control[0] = 75	Opcode
	control[1] = 4	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
reserved	int_in[0] =	Reserved for future use
object	int_in[1] =	The number (array index) of the object to watch
instate	int_in[2] =	The ob_state flag setting of the object when the mouse pointer (with button down) is inside of its rectangle
outstate	int_in[3] =	The ob_state flag setting of the object when the mouse pointer (with button down) is outside of its rectangle
tree	addr_in[0] =	The address of the object tree array which contains the object

Results

in_or_out	int_out[0] =	A code indicating the relative position of the mouse pointer when the user released the left mouse button: 0 = pointer is outside of object rectangle 1 = pointer is inside of object rectangle

See also

form_do( ), evnt_multi( ), evnt_button( )

Let the User Drag a Box Object

graf_slidebox( ) Opcode=76

This function draws a moving box outline which is the same size as a box object (slider) which is contained within a parent box object (slide bar). This moving box follows the mouse pointer, within the constraints of its container, so long as the user holds down the left mouse button. When the left mouse button is released, the function ends, and it returns a code indicating the relative position of the slider object within the slide bar object. It should only be called when the program has determined by returns from form_do( ), evnt_multi( ), or evnt_button( ), that the mouse pointer is positioned over the object and the left mouse button is down, since if the button is up when the call is made, the function will end immediately. This function can be used, for example, to implement a slide bar in a dialog, by making the slider a TOUCHEXIT object.

C binding

```
int status, parent, object, orientation;
OBJECT *tree;
   position = graf_slidebox(tree, parent, object, orientation);
```

Inputs

	control[0] = 76	Opcode
	control[1] = 3	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
parent	int_in[0] =	The number (array index) of the parent box object (slide bar)
object	int_in[1] =	The number (array index) of the child box object (slider)
orientation	int_in[2] =	A code indicating the orientation of the slide bar rectangle: 0 = horizontal (slider moves left-right) 1 = vertical (slider moves up-down)
tree	addr_in[0] =	The address of the object tree array which contains the objects

Results

position	int_out[0] =	A code indicating the position of the slider object relative to the parent slide bar. This code is a number in the range 0–1000, where, depending on orientation: 0 = left or top 1000 = right or bottom

Get the Physical Screen Handle

graf_handle( ) **Opcode=77**

This function returns the VDI handle number for the current physical screen workstation, along with some information about the default system font. The physical screen workstation is needed in order to open a virtual screen workstation with the v_opnvwk( ) call.

C binding

```
int phys_handle, cellw, cellh, boxw, boxh;
    phys_handle = graf_handle(&cellw, &cellh, &boxw, &boxh);
```

Inputs

control[0]	= 77	Opcode
control[1]	= 0	Number of 16-bit inputs in int_in array
control[2]	= 5	Number of 16-bit results in int_out array
control[3]	= 0	Number of 32-bit inputs in addr_in array
control[4]	= 0	Number of 32-bit results in addr_out array

Results

phys_handle	int_out[0] =	The VDI handle for the current physical screen workstation
cellw	int_out[1] =	The width, in pixels, of a character cell in the default system font (the one used to render text in menus and dialogs). The character cell is the entire space taken up by each character, including the intercharacter spacing.
cellh	int_out[2] =	The height, in pixels, of a character cell in the default system font (the one used to render text in menus and dialogs)
boxw	int_out[3] =	The width, in pixels, of a box surrounding a character cell in the default system font. Several GEM objects, such as vertical scroll bars, the close box, and size box are boxw pixels wide.
boxh	int_out[4] =	The height, in pixels, of a box surrounding a character cell in the default system font. Several GEM objects, such as horizontal scroll bars, the title bar, information line, close box, and size box are boxh pixels high.

Change the Mouse Pointer

graf_mouse( ) Opcode=78

Changes the shape of the mouse pointer. The caller may request one of eight predefined mouse pointer shapes or a user-defined shape made up of a 16 × 16-pixel bit image. This function may also be used to erase the mouse pointer before graphics operations are performed and to restore it after they are finished. This is necessary because the background behind the pointer is saved in a buffer and restored when the pointer is moved. If the mouse pointer isn't turned off during a drawing operation, the previous background image behind the pointer may be accidentally restored on top of the new background image created by the drawing operation.

If an application chooses to change the mouse pointer shape, it should only do so when the pointer is within the active (topmost) window. When the pointer leaves the active window, the program should change its shape back to an arrow or bee (shape 0 or 2). The application can use evnt_mouse( ) or evnt_multi( ) to track the movement of the mouse in and out of the active window rectangle, so as to know when to change mouse shape.

C binding

```
int status, form_no, formptr[37];
    status = graf_mouse(form_no, formptr);
```

Inputs

	control[0] = 78	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
form_no	int_in[0] =	A code indicating the new shape that the mouse pointer will assume. Valid code numbers include:

Form Number	Shape	Macro Name	Usage
0	Arrow	ARROW	General purposes
1	Vertical bar (I-beam)	TEXT_CRSR	Text cursor placement
2	Bee	HOURGLASS	Busy signal
3	Pointing hand	POINT_HAND	Sizing
4	Flat hand	FLAT_HAND	Dragging
5	Thin crosshairs	THIN_CROSS	Drawing
6	Thick crosshairs	THICK_CROSS	Application-specific
7	Outline crosshairs	OUTLN_CROSS	Application-specific
255	User defined	USER_DEF	Application-specific
256	Mouse pointer off	M_OFF	Hide mouse before drawing
257	Mouse pointer on	M_ON	Restore

formptr	addr_in[0] =	The address of a 37-word array that contains the data for the custom pointer shape. If one of the predefined shapes is requested, this value may be set to 0L. If a custom pointer is desired, the data array must first be set up, and its beginning address placed here. The format for this array is

Element Number	Description of Contents
0	X position of "hot spot"
1	Y position of "hot spot"
2	Number of bit planes (must be set to 1)
3	Background color (normally 0)
4	Foreground color (normally 1)
5–20	16 words of color mask data
21–36	16 words of image data

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Get Mouse and Shift-Key Status

graf_mkstate( ) Opcode=79

The graf_mkstate( ) call provides information about the current position of the mouse pointer on the screen and the current state of the mouse buttons and shift keys. Though this is the same kind of information returned by the evnt_button( ) and evnt_multi calls, the difference is that graf_mkstate( ) doesn't wait until an event occurs. Rather, it returns immediately, reporting the current status of the mouse buttons and shift keys. This makes it suitable for use in a polling routine that checks one or both of the mouse buttons.

Note that AES input functions like graf_mkstate( ) should never be mixed with VDI functions, or GEM will get very confused. Your program should use AES input functions or VDI input functions, but not both.

C binding

```
int reserved, mousex, mousey, mousbutn, shiftkey;
   reserved = graf_mkstate(&mousex, &mousey, &mousbutn, &shiftkey);
```

Inputs

control[0] = 79	Opcode
control[1] = 0	Number of 16-bit inputs in int_in array
control[2] = 5	Number of 16-bit results in int_out array
control[3] = 0	Number of 32-bit inputs in addr_in array
control[4] = 0	Number of 32-bit results in addr_out array

Results

reserved	int_out[0] =	Reserved for future use; always equals 1
mousex	int_out[1] =	The horizontal position of the mouse pointer at the end of the function
mousey	int_out[2] =	The vertical position of the mouse pointer at the end of the function
button	int_out[3] =	The final mouse button state: 0 = both buttons up 1 = left button down, right button up 2 = right button down, left button up 3 = both buttons down
shiftkey	int_out[4] =	The status of the keyboard shift keys. Each key is represented by a different bit. A 1 in that bit position means that the key is down, while a 0 means that it's up:

Bit	Bit Value	Key
0	1	Right Shift
1	2	Left Shift
2	4	Control
3	8	Alt

See also

evnt_button( ), evnt_multi( )

Read Scrap Directory

scrp_read( ) **Opcode=80**

By GEM convention, a disk may be used for a clipboard function to save data the user selects for a CUT or COPY operation. The program writes this data to disk in a file called SCRAP. This file may have any of several filename extensions (.TXT, .DIF, .IMG, and so on), depending on the type of data it contains, such as text, graphics, or spreadsheet data. So that other programs may share this data, when the program writes the file to disk, it gives the AES the pathname of the directory where the file resides, by using the scrp_write( ) function. When another program wishes to use that data, it finds the directory by using scrp_read( ).

C binding

```
int status;
char path[128];
    status = scrp_read(path);
```

Inputs

	control[0] = 80	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
	addr_in[0] =	The address of the buffer into which the scrap directory path will be written

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

scrp_write( )

Write Scrap Directory

scrp_write( ) **Opcode=81**

By GEM convention, a disk may be used for a clipboard function to save data the user selects for a CUT or COPY operation. The program writes this data to disk in a file called SCRAP. This file may have any of several filename extensions (.TXT, .DIF, .IMG, and so on), depending on the type of data it contains, such as text, graphics, or spreadsheet data. So that other programs may share this data, when the program writes the file to disk, it gives the AES the pathname of the directory where the file resides, by using the scrp_write( ) function. When another program wishes to use that data, it finds the directory by using scrp_read( ).

C binding

```
int status;
char path[128];
    status = scrp_write(path);
```

Inputs

	control[0] = 81	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
path	addr_in[0] =	Address of the buffer containing the text string which specifies the new scrap directory path

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

scrp_read( )

Display File Selector

fsel_input( ) **Opcode=90**

The File Selector is prepared dialog which displays a disk directory and obtains a pathname and filename from the user. The fsel_input( ) function displays this dialog, monitors the user's interaction with it, and returns the path and file names. Note that the directory display is limited to 100 files in any directory. After the user exits the file selector, the AES sends a redraw message to the application to repair the section of the window that was located under the selector dialog box. Your program should be prepared to handle that message by repairing the damage fsel_input( ) does to the display. Also you should note that fsel_input( ) changes the current VDI clipping rectangle and doesn't change it back upon exit from the routine. Therefore, if your program does any VDI rendering after a call to fsel_input( ), you'll probably have to set the clipping rectangle afterward, whether you normally use clipping or not.

C binding

```
int status, exitbutn;
char path[64], file[13];
    status = fsel_input(path, file, &exitbutn);
```

Inputs

	control[0] = 90	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 2	Number of 16-bit results in int_out array
	control[3] = 2	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
path	addr_in[0] =	The address of the buffer that holds the text string which specifies the path for the directory that's initially displayed in the dialog. This path name should use wildcards in the filename position (for instance, A: \UTILITIES *.* or C: \GAMES *.PRG). The call returns the user's final pathname selection in this same buffer.
file	addr_in[1] =	The address of the buffer that holds the text string which specifies the initial file selection that's displayed in the dialog box. The call returns the user's final filename selection in this same buffer.

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
exitbutn	int_out[1] =	A code which specifies the exit button which the user selected in order to end the dialog: 0 = Cancel 1 = OK

Allocate a Window

wind_create( ) **Opcode=100**

This function allocates the necessary resources for a window of a given maximum size having certain specified attributes. It returns a window handle that is used to identify the window. This function does not actually display the window on the screen, however. The wind_open( ) function is used for that purpose. Before the appl_exit( ) call is made to indicate that the application is about to terminate, all of the window resources should be released by using the wind_delete( ) call.

C binding

```
int wi_handle, controls, fullx, fully, fullw, fullh;

    wi_handle = wind_create(controls, fullx, fully, fullw, fullh);
```

Inputs

	control[0] = 100	Opcode
	control[1] = 5	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
controls	int_in[0] =	A code which specifies the window control components which will be present in this window. Each window control is represented by a bit in this word. If the bit which corresponds to a given control is set to 1, that control is present. The bit assignments are

Bit	Bit Value	Macro Name	Window Control
0	1 (0x001)	NAME	Title bar
1	2 (0x002)	CLOSER	Close box
2	4 (0x004)	FULLER	Full box
3	8 (0x008)	MOVER	Move bar
4	16 (0x010)	INFO	Information line
5	32 (0x020)	SIZER	Size box
6	64 (0x040)	UPARROW	Up arrow for vertical scroll bar
7	128 (0x080)	DNARROW	Down arrow for vertical scroll bar
8	256 (0x100)	VSLIDE	Slider for vertical scroll bar
9	512 (0x200)	LFARROW	Left arrow for horizontal scroll bar
10	1024 (0x400)	RTARROW	Right arrow for horizontal scroll bar
11	2048 (0x800)	HSLIDE	Slider for horizontal scroll bar

fullx	int_in[1] =	The screen position of the left edge of the maximum-size window
fully	int_in[2] =	The screen position of the top edge of the maximum-size window
fullw	int_in[3] =	The maximum width of the window, in pixels
fullh	int_in[4] =	The maximum height of the window, in pixels

Results

wi_handle	int_out[0] =	A unique number in the range 0–8 used to identify the window. Window handle number 0 is reserved for the Desktop window that's managed by the AES. If eight windows are already open (the maximum under the current version of GEM on the ST), a negative value will be returned, indicating that no windows were available, and the function failed to allocate a new window.

See also

wind_delete( ), wind_open

Display a Window

wind_open( ) **Opcode=101**

Displays a window in its initial size and position.

C binding

int status, wi_handle, x, y, width, height;
    status = wind_open(wi_handle, x, y, width, height);

Inputs

	control[0] = 101	Opcode
	control[1] = 5	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
wi_handle	int_in[0] =	The ID number of the window to be opened (returned initially by wind_create( )
x	int_in[1] =	The initial screen position of the window's left edge
y	int_in[2] =	The initial screen position of the window's top edge
width	int_in[3] =	The initial width of the window, in pixels
height	int_in[4] =	The initial height of the window, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_close( ), wind_create( )

Erase a Window

wind_close( ) Opcode=102

This call removes a window from the screen. Though the window is no longer displayed, its resources remain allocated, and it may be reopened at any time until the wind_delete( ) function is used to release those resources.

C binding

```
int wi_handle, status;
    status = wind_close(wi_handle);
```

Inputs

	control[0] = 102	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
wi_handle	int_in[0] =	The ID number of the window to be closed

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_open( ), wind_delete( )

Deallocate a Window

wind_delete( ) Opcode=103

This call is used to release the system resources held by a window. The wind_close( ) call should first be used to erase the window's screen display. Once a window's resources have been released, it cannot be opened again until the wind_create( ) call is used to allocate them again.

C binding

```
int wi_handle, status;
    status = wind_delete(wi_handle);
```

Inputs

	control[0] =	103	Opcode
	control[1] =	1	Number of 16-bit inputs in int_in array
	control[2] =	1	Number of 16-bit results in int_out array
	control[3] =	0	Number of 32-bit inputs in addr_in array
	control[4] =	0	Number of 32-bit results in addr_out array
wi_handle	int_in[0] =		The ID number of the window to be deleted

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_create( ), wind_close( )

Get Window Information

wind_get( ) Opcode=104

This function can be used to learn about the size and position of the window rectangle and its scroll bars.

C binding

```
int status, wi_handle, flag, x, y, width, height;
    status = wind_get(wi_handle, flag, &x, &y, &width, &height);
```

Inputs

	control[0] = 104	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 5	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
wi_handle	int_in[0] =	The ID number of the window about which information is to be provided
flag	int_in[1] =	A code indicating the type of information requested. Valid numbers include:

Flag	Macro Name	Information Requested
4	WF_WORKXYWH	Window work area coordinates
5	WF_CURRXYWH	Window exterior coordinates
6	WF_PREVXYWH	Previous window exterior coordinates
7	WF_FULLXYWH	Maximum window exterior coordinates
8	WF_HSLIDE	*x* = relative position of horizontal slider (1 = leftmost position, 1000 = rightmost)
9	WF_VSLIDE	*y* = relative position of vertical slider (1 = top position, 1000 = bottom)
10	WF_TOP	*x* = window handle of the top (active) window
11	WF_FIRSTXYWH	Coordinates of the first rectangle in the window's rectangle list
12	WF_NEXTXYWH	Coordinates of the next rectangle in the window's rectangle list
13	WF_RESVD	Reserved for future use
15	WF_HSLSIZE	*x* = relative size of the horizontal slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of a percent)
16	WF_VSLSIZE	*y* = relative size of the vertical slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of a percent)
17	WF_SCREEN	Address and length of the menu/alert buffers (*x* = low word of address, *y* = high word, width = low word of length, height = high word)

Results

Except where noted above in the description of the flag variable (int_in[1]), the four values returned specify the horizontal position, vertical position, width, and height of a screen rectangle. For functions 8, 9, 10, 15, and 16, the value in int_out[1]—associated with the variable name *x*—has some significance other than being a horizontal screen position value. For function number 17, the value in int_out[2] is used as the second part of an address. For specifics, see the descriptions of the various subfunctions.

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
x	int_out[1] =	The screen position of the left side of the window rectangle
y	int_out[2] =	The screen position of the top edge of the window rectangle
width	int_out[3] =	The width of the window rectangle, in pixels
height	int_out[4] =	The height of the window rectangle, in pixels

See also

wind_set( )

Change Window Settings

wind_set( ) **Opcode=105**

This function can be used to change a number of settings which affect the way in which a window is displayed.

C binding

int status, wi_handle, field, x, y, width, height;
    status = wind_set(wi_handle, field, x, y, width, height);

Inputs

	control[0] =	105	Opcode
	control[1] =	6	Number of 16-bit inputs in int_in array
	control[2] =	1	Number of 16-bit results in int_out array
	control[3] =	0	Number of 32-bit inputs in addr_in array
	control[4] =	0	Number of 32-bit results in addr_out array
wi_handle	int_in[0] =		The ID number of the window whose settings are to be changed
field	int_in[1] =		A code which specifies the window setting to change. Valid setting changes are

Field Number	Macro Name	Aspect to Change
1	WF_KIND	x = Window controls flag (same as controls for wind_create( ))
2	WF_NAME	x,y = Address of string containing the name of the window
3	WF_INFO	x,y = Address of string for the window's information line
5	WF_CURRXYWH	Window exterior coordinates
8	WF_HSLIDE	x = relative position of horizontal slider (1 = leftmost position, 1000 = rightmost)
9	WF_VSLIDE	x = relative position of vertical slider (1 = top position, 1000 = bottom)
10	WF_TOP	x = window handle of the top (active) window
14	WF_NEWDESK	The address of an object tree to be used for the Desktop Window background (x = low word, y = high word of address, width = number of starting object to draw)
15	WF_HSLSIZE	x = relative size of the horizontal slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of a percent)
16	WF_VSLSIZE	x = relative size of the vertical slider compared to the scroll bar (−1 = minimum size, 1–1000 = percentage filled, in tenths of a percent)

x	int_in[2] =	Specific to setting; as described above
y	int_in[3] =	Specific to setting; as described above
width	int_in[4] =	Specific to setting; as described above
height	int_in[5] =	Specific to setting; as described above

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

wind_get( )

Find Window Under Mouse Pointer

wind_find( ) **Opcode=106**

This function determines which window is currently under the mouse pointer, either the Desktop window (handle 0) or one of the application windows, whose handles are assigned by the wind_create( ) call.

C binding

```
int wi_handle, mousex, mousey;
    wi_handle = wind_find(mousex, mousey);
```

Inputs

	control[0] = 106	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
mousex	int_in[0] =	The horizontal screen position of the mouse pointer
mousey	int_in[1] =	The vertical screen position of the mouse pointer

Results

wi_handle	int_out[0] =	The ID number of the window located at the specified position

See also

objc_find( )

Lock or Release Screen for Update

wind_update( ) **Opcode=107**

This function is normally used to notify the AES prior to a screen update, so the AES won't change the screen display in that area (by dropping down a menu, for example). It can also be used by an application to take complete control of all mouse functions, even when the mouse is located outside of the active application window (in the menu bar, for example).

C binding

```
int status, code;
   status = wind_update(code);
```

Inputs

	control[0] =	107	Opcode
	control[1] =	1	Number of 16-bit inputs in int_in array
	control[2] =	1	Number of 16-bit results in int_out array
	control[3] =	0	Number of 32-bit inputs in addr_in array
	control[4] =	0	Number of 32-bit results in addr_out array
code	int_in[0] =		A code which specifies the function this call will perform. Valid code numbers are

Code	Macro Name	Function
0	BEG_UPDATE	Notifies AES that the application is beginning a window display update
1	END_UPDATE	Notifies AES that the application is ending its window display update
2	BEG_MCTRL	Notifies AES that the application is taking control of all mouse functions, even when it moves out of the active window
3	END_MCTRL	Notifies AES that it should once more take control of the mouse when it leaves the active window area

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Calculate Window Area

wind_calc( ) Opcode=108

Given the size and position of either the window's border rectangle or work rectangle, and its window control components, this function calculates the size and position of the opposite rectangle.

C binding

int status, type, controls, knownx, knowny, knownw, knownh, otherx, othery, otherw, otherh;
status = wind_calc (type, controls, knownx, knowny, knownw, knownh, &otherx, &othery, &otherw, &otherh);

Inputs

	control[0] = 108	Opcode
	control[1] = 6	Number of 16-bit inputs in int_in array
	control[2] = 5	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
type	int_in[0] =	A code specifying the type of calculation to perform: 0 = return border position and size 1 = return work position and size
controls	int_in[1] =	A code which specifies the window control components present in this window. Each window control is represented by a bit in this word. If the bit which corresponds to a given control is set to 1, that control is present. The bit assignments are

Bit	Bit Value	Macro Name	Window Control
0	1 (0x001)	NAME	Title Bar
1	2 (0x002)	CLOSER	Close Box
2	4 (0x004)	FULLER	Full Box
3	8 (0x008)	MOVER	Move Bar
4	16 (0x010)	INFO	Information Line
5	32 (0x020)	SIZER	Size Box
6	64 (0x040)	UPARROW	Up arrow for vertical scroll bar
7	128 (0x080)	DNARROW	Down arrow for vertical scroll bar
8	256 (0x100)	VSLIDE	Slider for vertical scroll bar
9	512 (0x200)	LFARROW	Left arrow for horizontal scroll bar
10	1024 (0x400)	RTARROW	Right arrow for horizontal scroll bar
11	2048 (0x800)	HSLIDE	Slider for horizontal scroll bar

knownx	int_in[2] =	The screen position of the left edge of the known rectangle
knowny	int_in[3] =	The screen position of the top edge of the known rectangle
knownw	int_in[4] =	The width of the known rectangle, in pixels

knownh	int_in[5] =	The height of the known rectangle, in pixels

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
otherx	int_out[1] =	The screen position of the left edge of the unknown rectangle
othery	int_out[2] =	The screen position of the top edge of the unknown rectangle
otherw	int_out[3] =	The width of the unknown rectangle, in pixels
otherh	int_out[4] =	The height of the unknown rectangle

Load a Resource

rsrc_load( ) Opcode=110

This function allocates memory for a resource file, loads it into memory, and performs the steps required to change the file into an object array of the proper format. These steps include changing array offsets into absolute addresses, and character-aligned screen references into absolute horizontal and vertical positions. The rsrc_load( ) function calls the routines rsrc_obfix( ) and rsrc_saddr( ) to perform these conversions. Once the resource has been loaded into memory, the address of a particular array element may be found using the rsrc_gaddr( ) function.

C binding

```
int status;
char *filename;
    status = rsrc_load(filename);
```

Inputs

	control[0] = 110	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
filename	addr_in[0] =	The address of a null-terminated text string which contains the path and filename of the resource file to load

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

rsrc_gaddr( ), rsrc_saddr( ), rsrc_obfix( )

Unload a Resource File

rsrc_free( ) Opcode=111

This function frees the memory space allocated for the resource file by the rsrc_load( ) call. It should be used by an application before loading a replacement resource file, and before calling appl_exit( ) to notify the AES that the application is about to terminate.

C binding

```
int status;
    status = rsrc_free( );
```

Inputs

	control[0] = 111	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

rsrc_load( )

Get Address of Resource Data

rsrc_gaddr( ) **Opcode=112**

This function is used to find the address of a data structure within a resource file that was loaded with rsrc_load( ).

C binding

```
int status, type, index;
long address;
    status = rsrc_gaddr(type, index, &address);
```

Inputs

	control[0] = 112	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 0	Number of 32-bit inputs in addr_in array
	control[4] = 1	Number of 32-bit results in addr_out array
type	int_in[0] =	A code specifying the type of structure containing the data whose address is to be returned. Valid code numbers are

Type Number	Macro Name	Structure
0	R_TREE	Object tree
1	R_OBJECT	OBJECT
2	R_TEDINFO	TEDINFO
3	R_ICONBLK	ICONBLK
4	R_BITBLK	BITBLK
5	R_STRING	Pointer to free strings
6	R_IMAGEDATA	Pointer to free image data
7	R_OBSPEC	Ob_spec field of OBJECT
8	R_TEPTEXT	Te_ptext field of TEDINFO
9	R_TEPTMPLT	Te_ptmplt field of TEDINFO
10	R_TEPVALID	Te_pvalid field of TEDINFO
11	R_IBPMASK	Ib_pmask field of ICONBLK
12	R_IBPDATA	Ib_pdata field of ICONBLK
13	R_IBPTEXT	Ib_ptext field of ICONBLK
14	R_BIPDATA	Bi_pdata field of BITBLK
15	R_FRSTR	Ad_frstr—the address of a pointer to a free string
16	R_FRIMG	Ad_frimg—the address of a pointer to a free image

index	int_in[1] =	The array index within the structure of the data whose address is sought

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution
address	addr_out[0] =	The address of the data whose position is specified by index and whose type is specified by the type variable.

See also

rsrc_load( )

Store Address of Resource Data

rsrc_saddr( ) **Opcode=113**

This function is used to store the address of a data structure within one of the elements of the arrays that was contained in the resource file that was loaded. It's used by rsrc_load( ) to fix the addresses of pointer fields like ob_spec, te_ptext, etc. It can also be used by an application, for example, to dynamically change the text field of a string object.

C binding

```
int status, type, index;
long address;
   status = rsrc_saddr(type, index, address);
```

Inputs

	control[0] = 113	Opcode
	control[1] = 2	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
type	int_in[0] =	A code specifying the type of structure into which the address will be placed. Valid code numbers are

Type Number	Macro Name	Data Structure
0	R_TREE	Object tree
1	R_OBJECT	OBJECT
2	R_TEDINFO	TEDINFO
3	R_ICONBLK	ICONBLK
4	R_BITBLK	BITBLK
5	R_STRING	Pointer to free strings
6	R_IMAGEDATA	Pointer to free image data
7	R_OBSPEC	Ob_spec field of OBJECT
8	R_TEPTEXT	Te_ptext field of TEDINFO
9	R_TEPTMPLT	Te_ptmplt field of TEDINFO
10	R_TEPVALID	Te_pvalid field of TEDINFO
11	R_IBPMASK	Ib_pmask field of ICONBLK
12	R_IBPDATA	Ib_pdata field of ICONBLK
13	R_IBPTEXT	Ib_ptext field of ICONBLK
14	R_BIPDATA	Bi_pdata field of BITBLK
15	R_FRSTR	Ad_frstr—the address of a pointer to a free string
16	R_FRIMG	Ad_frimg—the address of a pointer to a free image

index	int_in[1] =	The array index within the structure where the address will be placed
address	addr_in[0] =	The actual address that will be placed within the data structure

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

rsrc_load( ), rsrc_gaddr( )

Fix Object Location and Size

rsrc_obfix( ) **Opcode=114**

This function is used to change an object's location and size fields from the character-based coordinate system used in resource files to absolute pixels coordinates. It is called internally by rsrc_load( ).

C binding

int status, object;
struct object tree[ ];
    status = rsrc_obfix(tree, object);

Inputs

	control[0] = 114	Opcode
	control[1] = 1	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
object	int_in[0] =	The number (array index) of the objects whose coordinates are to be converted
tree	addr_in[0] =	The address of the object tree array which contains the object

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

rsrc_load( )

Find Invoking Program

shel_read( ) Opcode=120

This function may be used by a GEM application to discover how it was invoked, either from the GEM Desktop or from another application, and to read the command tail used when invoking the program. This allows the program to return control to the program which called it. The shel_read( ) function should be called after appl_init( ), but before rsrc_load( ).

C binding

```
int status;
char command[128], tail[128];
    status = shel_read(command, tail);
```

Inputs

	control[0] = 120	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 2	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
command	addr_in[0] =	The address of the buffer that will hold the command text string. This is the drive specification, directory, and filename of the application that invoked the current program. If the current program was loaded from the DESKTOP program, only a carriage-return character will be returned in this buffer.
tail	addr_in[1] =	The address of the buffer that will hold the command tail text string. This is the text that followed the name of this program in the command string used to start the program.

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

shel_write( )

Run Another Application

shel_write( ) **Opcode=121**

This function is used to select an application other than the GEM DESKTOP program to load and run after the program terminates. In other words, you may use this command to chain programs, so that when one terminates, the other begins directly, without first returning to the DESKTOP. Typically, this function is used in conjunction with OUTPUT.PRG, the program which sends graphics output to the printer. Since OUTPUT.PRG relies on the GDOS, which has not been officially released as of this writing, it's difficult to say how this function will be used on the ST version of GEM. Moreover, this function does not work reliably in the current (preblitter) version of the operating system ROMs. Some of the features of this command, such as the "exit GEM to DOS" feature, are obviously geared to the MS-DOS version of GEM and are not applicable to the ST version.

C binding

```
int status, exitgem, graphics, isgem;
char command[128], tail[128];
    status = shel_write(exitgem, graphics, isgem, command, tail);
```

Inputs

	control[0] = 121	Opcode
	control[1] = 3	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 2	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
exitgem	int_in[0] =	A code which indicates whether or not to exit GEM and return to the DOS prompt. Obviously inapplicable to the ST, where GEM is always present. 0 = exit GEM AES 1 = run another application
graphics	int_in[2] =	A code which indicates whether the next application to run is a graphics application. Inapplicable to the ST, which always uses a bitmapped display. 0 = not a graphics application 1 = a graphics application
isgem	int_in[3] =	A code which specifies whether or not the application to run is a GEM application 0 = not a GEM application 1 = a GEM application
command	addr_in[0] =	The address of the buffer that holds the command text string. This is the drive specification, directory, and filename of the application to invoke.
tail	addr_in[1] =	The address of the buffer that holds the command tail text string. This is the text

		that follows the name of this program in the command string used to start the program.

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

See also

shel-read( )

Search for Filename

shel_find( ) Opcode=124

This function is used to search for a filename. It first searches the current directory and then each directory in the current search path. If it finds the file, it returns its full pathname.

C binding

```
int status;
char pathname[128];
    status = shel_get(pathname);
```

Inputs

	control[0] = 124	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 1	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
pathname	addr_in[0] =	The address of a buffer which initially holds the text string of the name of the file to search for. Upon return from this function, this buffer will hold the entire pathname of the file (including drive specification and directories) if the function was able to find the file.

Results

status	int_out[0] =	Error status code: 0 = an error occurred during execution >0 = no error occurred during execution

Search for Environment String

shel_envrn( ) Opcode=125

This function searches the DOS environment for an environment string and returns the address of the byte following that string. This is another function which is more applicable to the MS-DOS version of GEM than the ST version.

C binding

```
int reserved;
char *textptr;
char estring[80];
   reserved = shel_envrn(&textptr, estring);
```

Inputs

	control[0] = 125	Opcode
	control[1] = 0	Number of 16-bit inputs in int_in array
	control[2] = 1	Number of 16-bit results in int_out array
	control[3] = 2	Number of 32-bit inputs in addr_in array
	control[4] = 0	Number of 32-bit results in addr_out array
txtptr	addr_in[0] =	The function returns a pointer here to the address in the environment that starts with the character following the requested string. If the requested string is not found, a value of 0 is returned.
estring	addr_in[1] =	The address of the buffer which holds the text of the environment string to search for (for instance, *PATH=*)

Results

reserved	int_out[0] =	Reserved for future use; always equals 1

Appendix B

Extended Keyboard Codes

The AES keyboard event functions (evnt_keybd and evnt_multi), return a two-byte value for every key pressed, rather than a simple one-byte ASCII code. The first byte of this keycode is generally a unique key identifier that refers to the physical key struck, regardless of shift-key combinations. The second byte is usually the ASCII value of the key combination, which does depend on the state of the shift keys (Shift, Control, and Alt). The following table shows the keycodes, as four-digit hexadecimal numbers, for all key and shift combinations.

Main Keyboard

Unshifted		Shift		CTRL	ALT
a	1E61	A	1E41	1E01	1E00
b	3062	B	3042	3002	3000
c	2E63	C	2E43	2E03	2E00
d	2064	D	2044	2004	2000
e	1265	E	1245	1205	1200
f	2166	F	2146	2106	2100
g	2267	G	2247	2207	2200
h	2368	H	2348	2308	2300
i	1769	I	1749	1709	1700
j	246A	J	244A	240A	2400
k	256B	K	254B	250B	2500
l	266C	L	264C	260C	2600
m	326D	M	324D	320D	3200
n	316E	N	314E	310E	3100
o	186F	O	184F	180F	1800
p	1970	P	1950	1910	1900
q	1071	Q	1051	1011	1000
r	1372	R	1352	1312	1300
s	1F73	S	1F53	1F13	1F00
t	1474	T	1454	1414	1400
u	1675	U	1655	1615	1600
v	2F76	V	2F56	2F16	2F00
w	1177	W	1157	1117	1100
x	2D78	X	2D58	2D18	2D00
y	1579	Y	1559	1519	1500
z	2C7A	Z	2C5A	2C1A	2C00

Unshifted		Shift		CTRL	ALT
1	0231	!	0221	0211	7800
2	0332	@	0340	0300	7900
3	0433	#	0423	0413	7A00
4	0534	$	0524	0514	7B00
5	0635	%	0625	0615	7C00
6	0736	^	075E	071E	7D00
7	0837	&	0826	0817	7E00
8	0938	*	092A	0918	7F00
9	0A39	(	0A28	0A19	8000
0	0B30	)	0B29	0B10	8100
-	0C2D	_	0C5F	0C1F	8200
=	0D3D	+	0D2B	0D1D	8300
`	2960	~	297E	2900	2960
\	2B5C	|	2B7C	2B1C	2B5C
[	1A5B	{	1A7B	1A1B	1A5B
]	1B5D	}	1B7D	1B1D	1B5D
;	273B	:	273A	271B	273B
'	2827	"	2822	2807	2827
,	332C	<	333C	330C	332C
.	342E	>	343E	340E	342E
/	352F	?	353F	350F	352F
Space	3920		3920	3900	3920
Esc	011B		011B	011B	011B
Backspace	0E08		0E08	0E08	0E08
Delete	537F		537F	531F	537F
Return	1C0D		1C0D	1C0A	1C0D
Tab	0F09		0F09	0F09	0F09

Cursor Pad

Unshifted		Shift	CTRL	ALT
Help	6200	6200	6200	(screen print)
Undo	6100	6100	6100	6100
Insert	5200	5230	5200	(left mouse button)
Clr/Home	4700	4737	7700	(right mouse button)
Up-Arrow	4800	4838	4800	(move mouse up)
Dn-Arrow	5000	5032	5000	(move mouse down)
Rt-Arrow	4B00	4B34	7300	(move mouse right)
Lft-Arrow	4D00	4D36	7400	(move mouse left)

Numeric Pad

Unshifted		Shift	CTRL	ALT
(	6328	6328	6308	6328
)	6429	6429	6409	6429
/	652F	652F	650F	652F
*	662A	662A	660A	662A
–	4A2D	4A2D	4A1F	4A2D
+	4E2B	4E2B	4E0B	4E2B
.	712E	712E	710E	712E
Enter	720D	720D	720A	720D
0	7030	7030	7010	7030
1	6D31	6D31	6D11	6D31
2	6E32	6E32	6E00	6E32
3	6F33	6F33	6F13	6F33
4	6A34	6A34	6A14	6A34
5	6B35	6B35	6B15	6B35
6	6C36	6C36	6C1E	6C36
7	6737	6737	6717	6737
8	6838	6838	6818	6838
9	6939	6939	6919	6939

Function Keys

Unshifted		Shift	CTRL	ALT
F1	3B00	5400	3B00	3B00
F2	3C00	5500	3C00	3C00
F3	3D00	5600	3D00	3D00
F4	3E00	5700	3E00	3E00
F5	3F00	5800	3F00	3F00
F6	4000	5900	4000	4000
F7	4100	5A00	4100	4100
F8	4200	5B00	4200	4200
F9	4300	5C00	4300	4300
F10	4400	5D00	4400	4400

Appendix C

Resource Files for Sample Programs

Four of the sample programs in this book require resource files in order to run. For those who have resource construction programs, there is a description of the required resource structure after the source code of each program, and, in the case of dialog boxes, there will be an illustration of the dialog as well. For those who don't have a resource construction program, the best advice is to obtain one as quickly as possible. In the meantime, however, this appendix presents an alternate method of creating the resource files.

The main program, RSCBUILD.C, merely writes a string of bytes to a disk file. The data it writes comes from an array called rscdata, which is part of a different file that's #included in RSCBUILD.C. The name of this file depends on which resource you wish to build. For example, in order to build DIALOG1.RSC, you would #include DIALOG1.DAT in the RSCBUILD.C program. To create MENU1.RSC, you would type in MENU1.DAT and make sure that file is #included in RSCBUILD.C. After you have compiled the program, execute it, and it will automatically create the resource file on disk. Be careful to type in all of the data for the rscdata array correctly, so that the resource structure will be properly recreated.

Program C-1. rscbuild.c

```
/*******************************************/
/*                                         */
/*   RSCBUILD.C                            */
/*   Builds resource files for dem         */
/*   #include the correct file             */
/*                                         */
/*******************************************/

#include <osbind.h>
#include "dialog1.dat"    /* substitute the name of the file */
                          /* you're using here */

main()
{

   int handle, error;

   handle = Fcreate(FILENAME,0);
   if (handle<0)
     {
     puts("can't open that file");
     exit(0);
     }
    error = Fwrite(handle, FILELEN, rscdata);
    printf (" We wrote %d bytes\n",error);
    Fclose(handle);
```

APPENDIX C

Program C-2. dialog1.dat

```
/*******************************************/
/*                                         */
/*   DIALOG1.DAT                           */
/*   Data for dialog1.rsc                  */
/*   #include with RSCBUILD.C              */
/*                                         */
/*******************************************/

#define FILENAME "DIALOG1.RSC"
#define FILELEN 1294L

int rscdata[647] =
    {
    0x0001, 0x01A6, 0x018A, 0x018A, 0x018A, 0x018A, 0x0024, 0x018A,
    0x018A, 0x0506, 0x0024, 0x0002, 0x0001, 0x0000, 0x0000, 0x0000,
    0x0000, 0x050E, 0x5F5F, 0x5F5F, 0x5F5F, 0x5F5F, 0x5F5F, 0x5F5F,
    0x004F, 0x7468, 0x6572, 0x3A20, 0x5F5F, 0x5F5F, 0x5F5F, 0x5F5F,
    0x5F5F, 0x5F5F, 0x006E, 0x6E6E, 0x6E6E, 0x6E6E, 0x6E6E, 0x6E6E,
    0x6E00, 0x434F, 0x4D50, 0x5554, 0x4552, 0x2053, 0x5552, 0x5645,
    0x5900, 0x556E, 0x6465, 0x7220, 0x3136, 0x0031, 0x362D, 0x3339,
    0x004F, 0x7665, 0x7220, 0x3339, 0x0041, 0x6765, 0x3A00, 0x436F,
    0x6D70, 0x7574, 0x6572, 0x7320, 0x4F77, 0x6E65, 0x643A, 0x0041,
    0x7461, 0x7269, 0x2053, 0x5400, 0x4174, 0x6172, 0x6920, 0x584C,
    0x2F58, 0x4500, 0x4578, 0x6964, 0x7920, 0x536F, 0x7263, 0x6572,
    0x6572, 0x0043, 0x616E, 0x6365, 0x6C00, 0x4F4B, 0x0020, 0x4465,
    0x736B, 0x2000, 0x2046, 0x696C, 0x6520, 0x0020, 0x2054, 0x6869,
    0x7320, 0x5370, 0x6163, 0x6520, 0x466F, 0x7220, 0x5265, 0x6E74,
    0x002D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x0020, 0x2044, 0x6573, 0x6B20,
    0x4163, 0x6365, 0x7373, 0x6F72, 0x7920, 0x3120, 0x2000, 0x2020,
    0x4465, 0x736B, 0x2041, 0x6363, 0x6573, 0x736F, 0x7279, 0x2032,
    0x2020, 0x0020, 0x2044, 0x6573, 0x6B20, 0x4163, 0x6365, 0x7373,
    0x6F72, 0x7920, 0x3320, 0x2000, 0x2020, 0x4465, 0x736B, 0x2041,
    0x6363, 0x6573, 0x736F, 0x7279, 0x2034, 0x2020, 0x0020, 0x2044,
    0x6573, 0x6B20, 0x4163, 0x6365, 0x7373, 0x6F72, 0x7920, 0x3520,
    0x2000, 0x2020, 0x4465, 0x736B, 0x2041, 0x6363, 0x6573, 0x736F,
    0x7279, 0x2036, 0x2020, 0x0020, 0x2053, 0x7572, 0x7665, 0x792E,
    0x2E2E, 0x0020, 0x2051, 0x7569, 0x7400, 0x0000, 0x0024, 0x0000,
    0x0031, 0x0000, 0x0045, 0x0003, 0x0006, 0x0000, 0x1180, 0x0000,
    0xFFFF, 0x000D, 0x0014, 0xFFFF, 0x0001, 0x0011, 0x0014, 0x0000,
    0x0010, 0x0002, 0x1100, 0x0003, 0x0000, 0x0027, 0x0813, 0x0002,
    0xFFFF, 0xFFFF, 0x001D, 0x0008, 0x0000, 0x0000, 0x018A, 0x000C,
    0x000E, 0x0013, 0x0001, 0x0003, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0052, 0x0010, 0x0001, 0x000F, 0x0001, 0x0007,
    0x0004, 0x0006, 0x0019, 0x0000, 0x0000, 0x0000, 0x1100, 0x0007,
    0x0004, 0x001B, 0x0001, 0x0005, 0xFFFF, 0xFFFF, 0x001A, 0x0011,
    0x0000, 0x0000, 0x0062, 0x0000, 0x0000, 0x0009, 0x0001, 0x0006,
    0xFFFF, 0xFFFF, 0x001A, 0x0011, 0x0000, 0x0000, 0x006B, 0x000A,
    0x0000, 0x0008, 0x0001, 0x0003, 0xFFFF, 0xFFFF, 0x001A, 0x0011,
    0x0000, 0x0000, 0x0071, 0x0013, 0x0000, 0x0008, 0x0001, 0x0008,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0079, 0x0002,
    0x0004, 0x0004, 0x0001, 0x0009, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x007E, 0x0002, 0x0006, 0x0010, 0x0001, 0x000A,
    0xFFFF, 0xFFFF, 0x0014, 0x0001, 0x0000, 0x00FF, 0x1101, 0x0007,
    0x0008, 0x0003, 0x0001, 0x000B, 0xFFFF, 0xFFFF, 0x0014, 0x0001,
    0x0000, 0x00FF, 0x1101, 0x0007, 0x000A, 0x0003, 0x0001, 0x000C,
    0xFFFF, 0xFFFF, 0x0014, 0x0001, 0x0000, 0x00FF, 0x1101, 0x0007,
    0x000C, 0x0003, 0x0001, 0x000D, 0xFFFF, 0xFFFF, 0x0014, 0x0001,
    0x0000, 0x00FF, 0x1101, 0x0007, 0x000E, 0x0003, 0x0001, 0x000E,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x008F, 0x000C,
    0x0008, 0x0008, 0x0001, 0x000F, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0098, 0x000C, 0x000A, 0x000B, 0x0001, 0x0010,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00A4, 0x000C,
    0x000C, 0x000E, 0x0001, 0x0011, 0xFFFF, 0xFFFF, 0x001A, 0x0005,
    0x0000, 0x0000, 0x00B3, 0x0004, 0x0011, 0x0008, 0x0001, 0x0000,
```

```
0xFFFF, 0xFFFF, 0x001A, 0x0027, 0x0000, 0x0000, 0x00BA, 0x001B,
0x0011, 0x0008, 0x0001, 0xFFFF, 0x0001, 0x0005, 0x0019, 0x0000,
0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0050, 0x0019, 0x0005,
0x0002, 0x0002, 0x0014, 0x0000, 0x0000, 0x0000, 0x1100, 0x0000,
0x0000, 0x0050, 0x0201, 0x0001, 0x0003, 0x0004, 0x0019, 0x0000,
0x0000, 0x0000, 0x0000, 0x0002, 0x0000, 0x000C, 0x0301, 0x0004,
0xFFFF, 0xFFFF, 0x0020, 0x0000, 0x0000, 0x0000, 0x00BD, 0x0000,
0x0000, 0x0006, 0x0301, 0x0002, 0xFFFF, 0xFFFF, 0x0020, 0x0000,
0x0000, 0x0000, 0x00C4, 0x0006, 0x0000, 0x0006, 0x0301, 0x0000,
0x0006, 0x000F, 0x0019, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
0x0301, 0x0050, 0x0013, 0x000F, 0x0007, 0x000E, 0x0014, 0x0000,
0x0000, 0x00FF, 0x1100, 0x0002, 0x0000, 0x0017, 0x0008, 0x0008,
0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00CB, 0x0000,
0x0000, 0x0017, 0x0001, 0x0009, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
0x0008, 0x0000, 0x00E1, 0x0000, 0x0001, 0x0017, 0x0001, 0x000A,
0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00F9, 0x0000,
0x0002, 0x0017, 0x0001, 0x000B, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
0x0000, 0x0000, 0x010E, 0x0000, 0x0003, 0x0017, 0x0001, 0x000C,
0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0123, 0x0000,
0x0004, 0x0017, 0x0001, 0x000D, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
0x0000, 0x0000, 0x0138, 0x0000, 0x0005, 0x0017, 0x0001, 0x000E,
0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x014D, 0x0000,
0x0006, 0x0017, 0x0001, 0x0006, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
0x0000, 0x0000, 0x0162, 0x0000, 0x0007, 0x0017, 0x0001, 0x0005,
0x0010, 0x0011, 0x0014, 0x0000, 0x0000, 0x00FF, 0x1100, 0x0008,
0x0000, 0x000D, 0x0002, 0x0011, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
0x0000, 0x0000, 0x0177, 0x0000, 0x0000, 0x000D, 0x0001, 0x000F,
0xFFFF, 0xFFFF, 0x001C, 0x0020, 0x0000, 0x0000, 0x0183, 0x0000,
0x0001, 0x000D, 0x0001, 0x0000, 0x01A6, 0x0000, 0x0356
};
```

Program 2a. dialog1.h

```
/* resource set indices for DIALOG1 */

#define DIALTREE 0      /* form/dialog */
#define OTHERTXT 1      /* FTEXT in tree DIALTREE */
#define YUNGBUTN 4      /* BUTTON in tree DIALTREE */
#define MIDBUTN  5      /* BUTTON in tree DIALTREE */
#define OLDBUTN  6      /* BUTTON in tree DIALTREE */
#define STBOX    9      /* BOX in tree DIALTREE */
#define XLBOX    10     /* BOX in tree DIALTREE */
#define EXIDYBOX 11     /* BOX in tree DIALTREE */
#define OTHERBOX 12     /* BOX in tree DIALTREE */
#define CANBUTN  16     /* BUTTON in tree DIALTREE */
#define OKBUTN   17     /* BUTTON in tree DIALTREE */

#define MENUTREE 1      /* menu tree */
#define SURVITEM 16     /* STRING in tree MENUTREE */
#define QUITITEM 17     /* STRING in tree MENUTREE */
```

Program C-3. dialog2.dat

```
/*********************************************/
/*                                           */
/*    DIALOG2.DAT                            */
/*    Data for dialog2.rsc                   */
/*    #include with RSCBUILD.C               */
/*                                           */
/*********************************************/

#define FILENAME "DIALOG2.RSC"
#define FILELEN 410L

int rscdata[205] =
    {
    0x0001, 0x0076, 0x005A, 0x0076, 0x005A, 0x005A, 0x0024, 0x005A,
    0x005A, 0x0196, 0x000C, 0x0001, 0x0001, 0x0000, 0x0000, 0x0000,
    0x0000, 0x019A, 0x4558, 0x4954, 0x0020, 0x2020, 0x3000, 0x0000,
    0x536C, 0x6964, 0x6572, 0x2050, 0x6F73, 0x6974, 0x696F, 0x6E3A,
    0x004F, 0x5054, 0x494F, 0x4E31, 0x004F, 0x5054, 0x494F, 0x4E32,
    0x004F, 0x5054, 0x494F, 0x4E33, 0x0050, 0x0000, 0x0029, 0x0000,
    0x002E, 0x0000, 0x002F, 0x0003, 0x0006, 0x0000, 0x1180, 0x0000,
    0xFFFF, 0x0005, 0x0001, 0xFFFF, 0x0001, 0x000B, 0x0014, 0x0000,
    0x0010, 0x0002, 0x1100, 0x0001, 0x0000, 0x0027, 0x0012, 0x0003,
    0x0002, 0x0002, 0x0014, 0x0000, 0x0000, 0x00FF, 0x1101, 0x0020,
    0x0003, 0x0002, 0x000A, 0x0001, 0xFFFF, 0xFFFF, 0x0014, 0x0040,
    0x0000, 0x0001, 0x1171, 0x0000, 0x0000, 0x0002, 0x0001, 0x0004,
    0xFFFF, 0xFFFF, 0x001A, 0x0007, 0x0000, 0x0000, 0x0024, 0x0012,
    0x000F, 0x0008, 0x0001, 0x0005, 0xFFFF, 0xFFFF, 0x0015, 0x0000,
    0x0000, 0x0000, 0x005A, 0x0019, 0x0003, 0x0004, 0x0001, 0x0006,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0030, 0x000D,
    0x0001, 0x0010, 0x0001, 0x0007, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0041, 0x000A, 0x0005, 0x0007, 0x0001, 0x0008,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0049, 0x000A,
    0x0008, 0x0007, 0x0001, 0x0009, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0051, 0x000A, 0x000B, 0x0007, 0x0001, 0x000A,
    0xFFFF, 0xFFFF, 0x0014, 0x0001, 0x0000, 0x00FF, 0x1101, 0x0003,
    0x0004, 0x0006, 0x0003, 0x000B, 0xFFFF, 0xFFFF, 0x0014, 0x0001,
    0x0000, 0x00FF, 0x1101, 0x0003, 0x0007, 0x0006, 0x0003, 0x0000,
    0xFFFF, 0xFFFF, 0x0014, 0x0021, 0x0000, 0x00FF, 0x1101, 0x0003,
    0x000A, 0x0006, 0x0003, 0x0000, 0x0076
    };
```

Program 3a. dialog2.h

```
/* resource set indices for DIALCG2 */

#define DIALTREE 0      /* form/dialog */
#define SLIDEBAR 1      /* BOX in tree DIALTREE */
#define SLIDER   2      /* BOX in tree DIALTREE */
#define EXITBUTN 3      /* BUTTON in tree DIALTREE */
#define NUMBER   4      /* TEXT in tree DIALTREE */
#define OPTION1  9      /* BOX in tree DIALTREE */
#define OPTION2  10     /* BOX in tree DIALTREE */
#define OPTION3  11     /* BOX in tree DIALTREE */
```

Program C-4. menu1.dat

```
/*******************************************/
/*                                         */
/*   MENU1.DAT                             */
/*   Data for menu1.rsc                    */
/*   #include with RSCBUILD.C              */
/*                                         */
/*******************************************/

#define FILENAME "MENU1.RSC"
#define FILELEN 1026L

int rscdata[513] =
    {
    0x0001, 0x01A6, 0x01A6, 0x01A6, 0x01A6, 0x01A2, 0x0024, 0x01A2,
    0x01A6, 0x03FE, 0x0019, 0x0001, 0x0000, 0x0000, 0x0000, 0x0001,
    0x0000, 0x0402, 0x2044, 0x6573, 0x6B20, 0x0020, 0x4669, 0x6C65,
    0x2000, 0x2044, 0x7261, 0x7720, 0x0020, 0x2041, 0x626F, 0x7574,
    0x204D, 0x656E, 0x7531, 0x2E2E, 0x2E00, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x0020, 0x2044, 0x6573, 0x6B20, 0x4163, 0x6365, 0x7373, 0x6F72,
    0x7920, 0x3120, 0x2000, 0x2020, 0x4465, 0x736B, 0x2041, 0x6363,
    0x6573, 0x736F, 0x7279, 0x2032, 0x2020, 0x0020, 0x2044, 0x6573,
    0x6B20, 0x4163, 0x6365, 0x7373, 0x6F72, 0x7920, 0x3320, 0x2000,
    0x2020, 0x4465, 0x736B, 0x2041, 0x6363, 0x6573, 0x736F, 0x7279,
    0x2034, 0x2020, 0x0020, 0x2044, 0x6573, 0x6B20, 0x4163, 0x6365,
    0x7373, 0x6F72, 0x7920, 0x3520, 0x2000, 0x2020, 0x4465, 0x736B,
    0x2041, 0x6363, 0x6573, 0x736F, 0x7279, 0x2036, 0x2020, 0x0020,
    0x2051, 0x7569, 0x7420, 0x205E, 0x5100, 0x2020, 0x5061, 0x7474,
    0x6572, 0x6E20, 0x3120, 0x5B46, 0x315D, 0x0020, 0x2050, 0x6174,
    0x7465, 0x726E, 0x2032, 0x205B, 0x4632, 0x5D00, 0x5B30, 0x5D5B,
    0x204D, 0x656E, 0x7520, 0x6465, 0x6D6F, 0x2077, 0x6974, 0x6820,
    0x6D75, 0x6C74, 0x692D, 0x6F62, 0x6A65, 0x6374, 0x7C20, 0x6974,
    0x656D, 0x7320, 0x616E, 0x6420, 0x6B65, 0x7962, 0x6F61, 0x7264,
    0x2065, 0x7175, 0x6976, 0x616C, 0x656E, 0x7473, 0x7C20, 0x2020,
    0x2020, 0x2053, 0x656C, 0x6563, 0x7420, 0x2251, 0x7569, 0x7422,
    0x2074, 0x6F20, 0x656E, 0x642E, 0x7C2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D5D, 0x5B49, 0x276C,
    0x6C20, 0x7265, 0x6D65, 0x6D62, 0x6572, 0x2074, 0x6861, 0x745D,
    0x0065, 0x0000, 0x010C, 0xFFFF, 0x0001, 0x0006, 0x0019, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0050, 0x0019, 0x0006,
    0x0002, 0x0002, 0x0014, 0x0000, 0x0000, 0x0000, 0x1100, 0x0000,
    0x0000, 0x0050, 0x0201, 0x0001, 0x0003, 0x0005, 0x0019, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0002, 0x0000, 0x0012, 0x0301, 0x0004,
    0xFFFF, 0xFFFF, 0x0020, 0x0000, 0x0000, 0x0000, 0x0024, 0x0000,
    0x0000, 0x0006, 0x0301, 0x0005, 0xFFFF, 0xFFFF, 0x0020, 0x0000,
    0x0000, 0x0000, 0x002B, 0x0006, 0x0000, 0x0006, 0x0301, 0x0002,
    0xFFFF, 0xFFFF, 0x0020, 0x0000, 0x0000, 0x0000, 0x0032, 0x000C,
    0x0000, 0x0006, 0x0301, 0x0000, 0x0007, 0x0012, 0x0019, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0301, 0x0050, 0x0013, 0x0010,
    0x0008, 0x000F, 0x0014, 0x0000, 0x0000, 0x00FF, 0x1100, 0x0002,
    0x0000, 0x0016, 0x0008, 0x0009, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0039, 0x0000, 0x0000, 0x0016, 0x0001, 0x000A,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0008, 0x0000, 0x004A, 0x0000,
    0x0001, 0x0016, 0x0001, 0x000B, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x0061, 0x0000, 0x0002, 0x0016, 0x0001, 0x000C,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0076, 0x0000,
    0x0003, 0x0016, 0x0001, 0x000D, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x00BB, 0x0000, 0x0004, 0x0016, 0x0001, 0x000E,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00A0, 0x0000,
    0x0005, 0x0016, 0x0001, 0x000F, 0xFFFF, 0xFFFF, 0x001C, 0x0000,
    0x0000, 0x0000, 0x00B5, 0x0000, 0x0006, 0x0016, 0x0001, 0x0007,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00CA, 0x0000,
    0x0007, 0x0016, 0x0001, 0x0012, 0x0011, 0x0011, 0x0014, 0x0000,
    0x0000, 0x00FF, 0x1100, 0x0008, 0x0000, 0x000C, 0x0001, 0x0010,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00DF, 0x0000,
    0x0000, 0x000C, 0x0001, 0x0006, 0x0013, 0x0018, 0x0014, 0x0000,
```

```
    0x0000, 0x00FF, 0x1000, 0x000E, 0x0000, 0x0015, 0x0002, 0x0014,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00EA, 0x0000,
    0x0000, 0x0010, 0x0001, 0x0015, 0xFFFF, 0xFFFF, 0x0014, 0x0000,
    0x0000, 0x00FF, 0x1162, 0x0010, 0x0000, 0x0005, 0x0001, 0x0016,
    0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00FB, 0x0000,
    0x0001, 0x0010, 0x0001, 0x0017, 0xFFFF, 0xFFFF, 0x0014, 0x0000,
    0x0000, 0x00FF, 0x1153, 0x0010, 0x0001, 0x0005, 0x0001, 0x0018,
    0xFFFF, 0xFFFF, 0x0019, 0x0000, 0x0000, 0x0000, 0x1100, 0x0000,
    0x0000, 0x0015, 0x0001, 0x0012, 0xFFFF, 0xFFFF, 0x0019, 0x0020,
    0x0000, 0x0000, 0x1100, 0x0000, 0x0001, 0x0015, 0x0001, 0x0000,
    0x01A6
    };
```

Program 4a. menu1.h

```
/* resource set indices for MENU1 */

#define MENUTREE 0      /* menu tree */
#define DESKTITL 3      /* TITLE in tree MENUTREE */
#define FILETITL 4      /* TITLE in tree MENUTREE */
#define DRAWTITL 5      /* TITLE in tree MENUTREE */
#define ABOTITEM 8      /* STRING in tree MENUTREE */
#define QUITITEM 17     /* STRING in tree MENUTREE */
#define DRAWPAT1 19     /* STRING in tree MENUTREE */
#define PAT1ITEM 23     /* IBOX in tree MENUTREE */
#define PAT2ITEM 24     /* IBOX in tree MENUTREE */

#define ABTALERT 0      /* Alert string index */
```

Program C-5. menu2.dat

```
/*********************************************/
/*                                           */
/*    MENU2.DAT                              */
/*    Data for menu2.rsc                     */
/*    #include with RSCBUILD.C               */
/*                                           */
/*********************************************/

#define FILENAME "MENU2.RSC"
#define FILELEN 1068L

int rscdata[534] =
    {
    0x0001, 0x0200, 0x0200, 0x0200, 0x0200, 0x01F4, 0x0024, 0x01F4,
    0x0200, 0x0428, 0x0017, 0x0001, 0x0000, 0x0000, 0x0000, 0x0003,
    0x0000, 0x042C, 0x2044, 0x6573, 0x6B20, 0x0020, 0x4669, 0x6C65,
    0x2000, 0x204F, 0x7074, 0x696F, 0x6E73, 0x2000, 0x2020, 0x4162,
    0x6F75, 0x7420, 0x4D65, 0x6E75, 0x322E, 0x2E2E, 0x002D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D00, 0x2020, 0x4465, 0x736B, 0x2041, 0x6363, 0x6573,
    0x736F, 0x7279, 0x2031, 0x2020, 0x0020, 0x2044, 0x6573, 0x6B20,
    0x4163, 0x6365, 0x7373, 0x6F72, 0x7920, 0x3220, 0x2000, 0x2020,
    0x4465, 0x736B, 0x2041, 0x6363, 0x6573, 0x736F, 0x7279, 0x2033,
    0x2020, 0x0020, 0x2044, 0x6573, 0x6B20, 0x4163, 0x6365, 0x7373,
    0x6F72, 0x7920, 0x3420, 0x2000, 0x2020, 0x4465, 0x736B, 0x2041,
    0x6363, 0x6573, 0x736F, 0x7279, 0x2035, 0x2020, 0x0020, 0x2044,
    0x6573, 0x6B20, 0x4163, 0x6365, 0x7373, 0x6F72, 0x7920, 0x3620,
    0x2000, 0x2020, 0x5175, 0x6974, 0x2020, 0x5E51, 0x0020, 0x2043,
    0x6865, 0x636B, 0x204D, 0x6172, 0x6B00, 0x2D2D, 0x2D2D, 0x2D2D,
    0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
    0x0020, 0x2054, 0x7572, 0x6E20, 0x4E65, 0x7874, 0x2049, 0x7465,
    0x6D20, 0x4F4E, 0x2000, 0x2020, 0x5072, 0x696E, 0x7420, 0x4D65,
    0x7373, 0x6167, 0x6500, 0x2020, 0x5475, 0x726E, 0x204E, 0x6578,
    0x7420, 0x4974, 0x656D, 0x204F, 0x4646, 0x005B, 0x305D, 0x5B20,
    0x4D65, 0x6E75, 0x2064, 0x656D, 0x6F20, 0x7769, 0x7468, 0x2063,
```

```
0x6865, 0x636B, 0x206D, 0x6172, 0x6B73, 0x2C7C, 0x2067, 0x7261,
0x7969, 0x6E67, 0x2C20, 0x616E, 0x6420, 0x616C, 0x7465, 0x726E,
0x6174, 0x6520, 0x7465, 0x7874, 0x2E7C, 0x2020, 0x2020, 0x2020,
0x5365, 0x6C65, 0x6374, 0x2022, 0x5175, 0x6974, 0x2220, 0x746F,
0x2065, 0x6E64, 0x2E7C, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D,
0x2D2D, 0x2D2D, 0x2D2D, 0x2D2D, 0x5D5B, 0x4927, 0x6C6C, 0x2072,
0x656D, 0x656D, 0x6265, 0x7220, 0x7468, 0x6174, 0x5D00, 0x2020,
0x5475, 0x726E, 0x204E, 0x6578, 0x7420, 0x4974, 0x656D, 0x204F,
0x4E20, 0x0000, 0x0000, 0x0136, 0x0000, 0x014B, 0x0000, 0x01DE,
0xFFFF, 0x0001, 0x0006, 0x0019, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0000, 0x0050, 0x0019, 0x0006, 0x0002, 0x0002, 0x0014,
0x0000, 0x0000, 0x0000, 0x1100, 0x0000, 0x0000, 0x0050, 0x0201,
0x0001, 0x0003, 0x0005, 0x0019, 0x0000, 0x0000, 0x0000, 0x0000,
0x0002, 0x0000, 0x0016, 0x0301, 0x0004, 0xFFFF, 0xFFFF, 0x0020,
0x0000, 0x0000, 0x0000, 0x0024, 0x0000, 0x0000, 0x0006, 0x0301,
0x0005, 0xFFFF, 0xFFFF, 0x0020, 0x0000, 0x0000, 0x0000, 0x002B,
0x0006, 0x0000, 0x0006, 0x0301, 0x0002, 0xFFFF, 0xFFFF, 0x0020,
0x0000, 0x0000, 0x0000, 0x0032, 0x000C, 0x0000, 0x000A, 0x0301,
0x0000, 0x0007, 0x0012, 0x0019, 0x0000, 0x0000, 0x0000, 0x0000,
0x0000, 0x0301, 0x0050, 0x0013, 0x0010, 0x0008, 0x000F, 0x0014,
0x0000, 0x0000, 0x00FF, 0x1100, 0x0002, 0x0000, 0x0016, 0x0008,
0x0009, 0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x003C,
0x0000, 0x0000, 0x0016, 0x0001, 0x000A, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0008, 0x0000, 0x004D, 0x0000, 0x0001, 0x0016, 0x0001,
0x000B, 0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x0064,
0x0000, 0x0002, 0x0016, 0x0001, 0x000C, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x0079, 0x0000, 0x0003, 0x0016, 0x0001,
0x000D, 0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x008E,
0x0000, 0x0004, 0x0016, 0x0001, 0x000E, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x00A3, 0x0000, 0x0005, 0x0016, 0x0001,
0x000F, 0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0000, 0x0000, 0x00B8,
0x0000, 0x0006, 0x0016, 0x0001, 0x0007, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x00CD, 0x0000, 0x0007, 0x0016, 0x0001,
0x0012, 0x0011, 0x0011, 0x0014, 0x0000, 0x0000, 0x00FF, 0x1100,
0x0008, 0x0000, 0x000C, 0x0001, 0x0010, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x00E2, 0x0000, 0x0000, 0x000C, 0x0001,
0x0006, 0x0013, 0x0016, 0x0014, 0x0000, 0x0000, 0x00FF, 0x1000,
0x000E, 0x0000, 0x0016, 0x0004, 0x0014, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x00ED, 0x0000, 0x0000, 0x0016, 0x0001,
0x0015, 0xFFFF, 0xFFFF, 0x001C, 0x0000, 0x0008, 0x0000, 0x00FA,
0x0000, 0x0001, 0x0016, 0x0001, 0x0016, 0xFFFF, 0xFFFF, 0x001C,
0x0000, 0x0000, 0x0000, 0x0111, 0x0000, 0x0002, 0x0016, 0x0001,
0x0012, 0xFFFF, 0xFFFF, 0x001C, 0x0020, 0x0008, 0x0000, 0x0126,
0x0000, 0x0003, 0x0016, 0x0001, 0x0000, 0x0200
};
```

Program 5a. menu2.h

```
/* resource set indices for MENU2 */

#define MENUTREE 0      /* menu tree */
#define DESKTITL 3      /* TITLE in tree MENUTREE */
#define FILETITL 4      /* TITLE in tree MENUTREE */
#define OPTNTITL 5      /* TITLE in tree MENUTREE */
#define ABOTITEM 8      /* STRING in tree MENUTREE */
#define QUITITEM 17     /* STRING in tree MENUTREE */
#define CHEKITEM 19     /* STRING in tree MENUTREE */
#define TOGLITEM 21     /* STRING in tree MENUTREE */
#define ABLEITEM 22     /* STRING in tree MENUTREE */

#define OFFSTRNG 0      /* Free string index */

#define ABTALERT 1      /* Alert string index */

#define ONSTRNG  2      /* Free string index */
```

Function Index

Function	Opcode Number	Page
wind_create( )	100	26–28, 35, 284–85
wind_open( )	101	27, 31, 33, 286
wind_close( )	102	34–35, 287
wind_delete( )	103	35, 288
wind_get( )	104	29, 58, 289–90
wind_set( )	105	31–33, 291–92
wind_find( )	106	293
wind_update( )	107	57, 61, 294
wind_calc( )	108	34, 65, 295–96
rsrc_load( )	110	135–36, 161, 297
rsrc_free( )	111	136, 298
rsrc_gaddr( )	112	136–37, 158, 161, 299–300
rsrc_saddr( )	113	138–39, 301–2
rsrc_obfix( )	114	138, 303
shel_read( )	120	304
shel_write( )	121	305–6
shel_find( )	124	307
shel_envrn( )	125	308

Index

The Complete GEM AES Reference

Intermediate to advanced Atari ST programmers will welcome this exhaustive reference to the ST's friendly interface—the Application Environment Services. *COMPUTE!'s Technical Reference Guide—Atari ST, Volume Two: The GEM AES* gives you complete information on starting an application, a two-part explanation of the use of windows, and details about GEM graphics objects, resource files, and desk accessories, to name only a few of the topics covered.

This is the second book in a series of three on the Atari ST. The first, concerned the Virtual Device Interface (VDI). *COMPUTE!'s Technical Reference Guide—Atari ST, Volume Two: The GEM AES* takes you further into the underpinnings of the GEM interface.

Here is a list of only a few of the ST features covered in this book:

- Menus, alert boxes, and dialog boxes.
- Input/output through radio buttons, mouse operation, and boxes.
- Sample programs written in C, machine language, and BASIC.
- Resource-development programs.
- A complete AES (Application Environment Services) function reference.
- Full cross-referencing of functions by name and opcode.

Sheldon Leemon has a long-standing reputation for accuracy, readability, and the assurance of expertise. In this book, he continues his investigation into the inner workings of one of the most versatile and popular computers on the market—the Atari ST.

COMPUTE!'s Technical Reference Guide—Atari ST, Volume Two: The GEM AES is the complete reference guide to the AES, the heart of the ST's friendly interface.

ISBN 0-87455-114-5

51895

9 780874 551143