– CHAPTER 6 –

AES

Overview

The Application Environment Services (**AES**) compose the highest level of the operating system. The **AES** uses the **VDI**, **GEMDOS**, and **XBIOS** to provide a global utility library of calls which provide applications with the **GEM** interface. Usage of the **AES** makes application development simpler and makes user interfaces more consistent. The services provided by the **AES** include:

- Application Control/Interaction
- Event Management
- Menu Services
- Object Rendering/Manipulation
- Form Management
- Graphic Utility Functions
- Scrap (Clipboard) Management
- Common Dialog Display
- Window Management
- Resource Management
- Shell (Desktop) Interaction

System-specific **AES** information and variables may be determined through reserved fields in the application's global array (see **appl_init()**) or by using the various modes of **appl_getinfo()**.

Process Handling

The **AES** manages two types of user programs. Normal **GEM** applications have file extensions of '.PRG', '.APP', or '.GTP'. Desk Accessories have file extensions of '.ACC'.

Without **MultiTOS**, the **AES** can have a maximum of one application and six desk accessories (four desk accessories under **TOS** 1.0) executing concurrently. The currently running application (or the Desktop if no application is running) is given primary control over the system. Desk accessories are allocated processor time *only* when the foreground application releases control by calling one of the event library functions. An application which does not have a standard event loop (as illustrated below) will cause desk accessories to stop functioning while it is being executed.

Under **MultiTOS**, an unlimited amount of applications and desk accessories may be loaded concurrently[1]. **MultiTOS** is a pre-emptive system where all system processes are given time regardless of other applications.

Applications

When an application is launched, **GEM** allocates *all* remaining system memory and loads the application into this area[2]. It is the responsibility of the application to free whatever memory it doesn't immediately need for its text, data, bss, and stack area. Most high level languages do this for you in the startup stub linked with every application.

GEM applications begin with an **appl_init()** function call. This call will return a valid application ID if the process can be successfully registered or a -1 if it fails. If the call fails, the application should immediately exit without making any **AES** calls. Upon success, however, the ID should be stored for future use within the application. Applications running under **MultiTOS** should call **menu_register()** to display the program title in the application list rather than the filename.

The next steps a **GEM** application will follow are variable, however, most **GEM** applications will initialize themselves further by performing some or all of the following steps:

- Open a **VDI** workstation.
- Verify that the computer the application is being run on has the minimum requirements (screen resolution, OS versions, memory needs, hardware features) necessary to continue.
- Load the application '.RSC' file and fix it up as necessary.
- Display the menu bar.
- Change the mouse form to an arrow (the **AES** defaults to a **BUSY_BEE** shape).
- Enter the application's main event loop.

The following represents a basic skeleton for an **AES** application:

```
#include <AES.H>
#include <VDI.H>
#include <OSBIND.H>
#include <VDIWORK.H>
#include "skel.h"

#define CNTRL_Q  0x11
```

[1] Some **MultiTOS** versions limit this based upon the available space in the leftmost menu.

[2] **TOS** 5.0 does allow the user to set limits on the amount of memory allowed to an application.

```
int main(int, char *[]);

extern int _AESglobal[15];

short ap_id;
VDI_Workstation ws;       /* See entry for V_Opnvwk() in VDI docs */
OBJECT *mainmenu;

char RSCname[] = "skeleton.rsc";
char menu_title[] = "  Skeleton";

int
main(int argc, char *argv[])
{
       char *altNoVDIWork = "[3][GEM is unable to|allocate a workstation.|The
       program       must abort.][ OK ]";
       char *altNoRSC = "[3][The program cannot locate|SKELETON.RSC. Please
             ensure|that it resides in the|same directory as|SKELETON.PRG.][ OK ]";
       short ret,msg[8],kc,quit,dum;

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

       if(!OpenVwork(&ws))
       {
             graf_mouse( ARROW, 0L );
             form_alert(1, altNoVDIWork );
             appl_exit();
             return -1;
       }

       if(!rsrc_load( RSCname ))
       {
             graf_mouse( ARROW, 0L );
             form_alert(1, altNoRSC );
             v_clsvwk(ws.handle);
             appl_exit();
             return -1;
       }

       if(_AESglobal[1] == -1)                          /* MultiTOS present?
       */
             menu_register(ap_id, menu_title);      /* Yes, make name pretty. */

       rsrc_gaddr( R_TREE, MAINMENU, &mainmenu);

       menu_bar(mainmenu,1);
       graf_mouse( ARROW, 0L );

       quit = FALSE;
       while(!quit)
       {
             ret = evnt_multi(MU_MESAG|MU_KEYBD,2,1,1,0,0,0,0,0,0,0,0,0,0,msg,0,0,
                              &dum,&dum,&dum,&dum,&kc,&dum);

             if(ret & MU_MESAG)
             {
```

```
                switch(msg[0])
                {
                    case MN_SELECTED:
                        switch(msg[3])
                        {
                            .       /* Other menu selections */
                            .
                            .

                            case mmExit:                  /* Defined in SKEL.H */
                                quit = TRUE;
                                break;
                        }
                        break;
                    }
                }
            }

            if(ret & MU_KEYBD)
            {
                switch(kc & 0xFF)
                {
                    .           /* Other keyboard equivalents */
                    .
                    .
                    case CNTRL_Q:
                        quit = TRUE;
                        break;
                }
            }
        }

        menu_bar( mainmenu, 0 );
        v_clsvwk(ws.handle);
        rsrc_free();
        appl_exit();
        return 0;
}
```

The Command Line

GEM applications, like **TOS** applications, may be started with a command line (for a detailed description of command line processing, see *Chapter 2:* ***GEMDOS***). '.PRG' files and '.APP' files will have items on the command line if a document file which was registered with the application was double-clicked or if a valid document file was dropped over the application's icon in the Desktop. Launching a '.GTP' application will cause the Desktop to prompt the user for a command line in the same manner as '.TTP' programs are handled. Applications which find one or more valid document names on their command line should automatically load them on program start.

Desk Accessories

Upon bootup, any files with the extension '.ACC' found in the root directory of the user's boot drive will be loaded and executed up until their first event library call. **MultiTOS** allows desk accessories to be loaded and unloaded after bootup.

Unlike applications, desk accessories are not given all of available system memory on startup. They are only allocated enough memory for their text, data, and bss segments. No stack space is allocated for a desk accessory either. Many high level language stubs reserve space in the BSS or overwrite startup code to provide a stack but keep in mind that desk accessory stacks are usually small compared to applications.

As with applications, **GEM** desk accessories should begin with an **appl_init()** function call. Upon success, the ID should be stored and used within a **menu_register()** call to place the applications' name on the menu bar.

Desk accessories, unlike applications, do not begin user interaction immediately. Most desk accessories initialize themselves and enter a message loop waiting for an **AC_OPEN** message. Some desk accessories wait for timer events or custom messages from another application. After being triggered, they usually open a window in which user interaction may be performed (dialogs and alerts may also be presented but are not recommended because they prevent shuffling between other system processes).

Desk accessories should not use a menu bar and should never exit (unless **appl_init()** fails) after calling **menu_register()**. If an error condition occurs which would make the accessory unusable, simply enter an indefinite message loop.

Any resources loaded by an accessory should be loaded prior to entering the first event loop and should never be freed after the accessory has called **menu_register()**. Resource data for desk accessories should be embedded in the executable rather than being soft-loaded because memory allocated to a desk accessory is not freed during a resolution change on **TOS** versions less than 2.06. This causes resource memory allocated by **rsrc_load()** to be lost to the system after a resolution change and will likely cause memory fragmentation.

An **AC_CLOSE** message is sent to an accessory when it is being closed at the request of the OS. At this point, it should perform any cleanup necessary to release system resources and close files opened at **AC_OPEN** (accessory windows will be closed automatically by the **AES**). After cleanup, the event loop should be reentered to wait for subsequent **AC_OPEN** messages.

The following code represents a basic skeleton for an **AES** desk accessory:

```
#include <AES.H>
#include <VDI.H>
#include <OSBIND.H>
#include <VDIWORK.H>
```

```
int main(int, char *[]);

short ap_id;
VDI_Workstation ws;       /* See entry for V_Opnvwk() in VDI docs */

char menu_title[] = "  Skeleton";

int
main(int argc, char *argv[])
{
       char *altNoVDIWork = "[3][GEM is unable to|allocate a workstation.|The
       program       must abort.][ OK ]";
       short ret,msg[8],kc,dum;

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

       if(!OpenVwork(&ws))
       {
        form_alert(1, altNoVDIWork);
            appl_exit();
            return -1;
       }

       menu_id = menu_register(ap_id, menu_title );     /* Place name on menu bar
       */

       for(;;)
       {
            evnt_mesag(msg);

            switch( msg[0] )
            {
                case AC_OPEN:
                    if(msg[3] == menu_id)
                        OpenAccessoryWindow();
                    break;
                case AC_CLOSE:
                    if(msg[3] == menu_id)
                    {
                        v_clsvwk(ws.handle);
                        break;
                    }
            }
        }
}
```

The Environment String

One **AES** environment string exists in the system. This environment string is the one initially allocated for the **AES** by **GEMDOS**. The **AES** environment string should not be confused with **GEMDOS** environment strings. Each **GEMDOS** process receives its own environment string when launched. This string may have been purposely altered (or omitted) by its parent.

The **AES** environment string is a collection of variables which the AES (and other processes) may use as global system variables. Environment data may be set by a CPX designed to configure the environment, in the user's GEM.CNF file, or by an application.

In actuality, the environment string is actually one or many string entries separated by **NULL** bytes with the full list being terminated by a double **NULL** byte. Examples of environment string entries include:

```
PATH=C:\;D:\;E:\BIN\
TEMP=C:\
AE_SREDRAW=0
```

The environment variable name is followed by an equal sign which is followed by the variable data. Multiple arguments (such as path names) may be separated by semicolons or commas[3].

The **AES** call **shel_envrn()** may be used to search for an environment variable and new modes of **shel_write()** (after **AES** version 4.0) may be used to alter environment variables or copy the entire environment string.

Most versions of the **AES** contain a bug which causes the 'PATH' environment variable to be set incorrectly upon bootup to 'PATH=*[nul]*A:*[nul][nul]*'. If an environment string like this is found it may be safely reset or simply ignored.

The Event Dispatcher

Most **GEM** applications and all desk accessories rely on one of the **AES** event processing calls to direct program flow. After program initialization, an application enters a message loop which waits for and reacts to messages sent by the **AES**. Five basic types of events are generated by the **AES** and each can be read by a specialized event library call as follows:

Event Type	AES Function
Message	**evnt_mesag()**
Mouse Button	**evnt_button()**
Keyboard	**evnt_keybd()**
Timer	**evnt_timer()**

[3]The **AES** only began recognizing commas as valid argument separators (for the PATH environment variable) as of **AES** version 1.4.

Mouse Movement	**evnt_mouse()**

In addition to these five basic calls, the **AES** offers one multi-purpose call which waits for any combination of the above events called **evnt_multi()**. The **evnt_multi()** call is often the most important function call in any **GEM** application. A typical message loop follows:

```
#include <AES.H>

void
MessageLoop( void )
{
       short mx, my;          /* Mouse Position */
       short mb, mc;          /* Mouse button/# clicks */
       short ks, kc;          /* Key state/code */
       short quit;            /* Exit flag */
       short msg[8];          /* Message buffer */
       short events;          /* What events are valid? */

/* Mask for all events */
#define ALL_EVENTS   (MU_MESAG|MU_BUTTON|MU_KEYBD|MU_TIMER|MU_M1|MU_M2)

       quit = FALSE;
       while(!quit)
       {
            events = evnt_multi( ALL_EVENTS,
                                 2, 1, 1,             /* Single/double clicks */
                                 0, 0, 0, 128, 128,   /* M1 event */
                                 1, 0, 0, 128, 128,   /* M2 event */
                                 msg,                 /* Pointer to msg */
                                 1000, 0,             /* MU_TIMER every 1 sec. */
                                 &mx, &my, &ks, &kc,
                                 &mc );

            if( events & MU_MESAG )
            {
                switch( msg[0] )     /* msg[0] is message type */
                {
                    case MN_SELECTED:
                        HandleMenuClick( msg );
                        break;
                    case WM_CLOSED:
                        CloseWindow( msg[3] );
                        break;
                    /*
                     * more message events...
                     */
                }
            }

            if( events & MU_BUTTON )
            {
                /*
                 * Handle mouse button event.
                 */
            }

            if( events & MU_KEYBD )
```

```
            {
                /*
                 * Handle keyboard events.
                 */
            }

            if( events & MU_TIMER )
            {
                /*
                 * Handle Timer events.
                 */
            }

            if( events & MU_M1 )
            {
                /*
                 * Handle mouse rectangle event 1.
                 */
            }

            if( events & MU_M2 )
            {
                /*
                 * Handle mouse rectangle event 2.
                 */
            }
        }

        /* Loop will terminate here when 'quit' is set to TRUE. */
}
```

When an event library function is called, the program is effectively halted until a message which is being waited for becomes available. Not all applications will require all events so the above code may be considered flexible.

Message Events

Each standard **GEM** message event (**MU_MESAG**) uses some or all of an 8 **WORD** message buffer. Each entry in this buffer is assigned as follows:

msg[x]	Meaning
0	Message type.
1	The application identifier of the process sending the message.
2	The length of the message *beyond* 16 bytes (in bytes). For all standard **GEM** messages, this values is 0.
3	Depends on message.
4	Depends on message.
5	Depends on message.
6	Depends on message.
7	Depends on message.

The entry for **evnt_mesag()** later in this chapter has a comprehensive list of all system messages and the action that should be taken when they are received.

User-Defined Message Events

Applications may write customized messages to other applications (or themselves) using **appl_write()**. The structure of the message buffer should remain the same as shown above. If more than the standard eight **WORD**s of data are sent, however, **appl_read()** must be used to read the additional bytes. It is recommended that user-defined messages be set to a multiple of 8 bytes.

You can use this method to send your own application standard messages by filling in the message buffer appropriately and using **appl_write()**. This method is often used to force redraw or window events.

Mouse Button Events

When a mouse button (**MU_BUTTON**) event happens, the **evnt_button()** or **evnt_multi()** call is returned with the mouse coordinates, the number of clicks that occurred, and the keyboard shift state.

Keyboard Events

Keyboard events (**MU_KEYBD**) are generated whenever a key is struck. The IKBD scan code (see *Appendix F: **IKBD Scan Codes***) and current key shift state are returned by either **evnt_keybd()** or **evnt_multi()**. If your application is designed to run on machines in other countries, you might consider translating the scan codes using the tables returned by the **XBIOS** call **Keytbl()**.

Timer Events

evnt_timer() or **evnt_multi(MU_TIMER**, ... **)** can be used to request a timer event(s) be scheduled in a certain number of milliseconds. The time between the actual function call and the event may, however, be greater than the time specified.

Mouse Rectangle Events

Mouse rectangle events (**MU_M1** and/or **MU_M2**) are generated by **evnt_mouse()** and **evnt_multi()** when the mouse pointer enters or leaves (depending on how you program it) a specified rectangle.

Resources

GEM resources consist of object trees, strings, and bitmaps used by an application. They encapsulate the user interface and make internationalization easier by placing all program strings in a single file. Resources are generally created using a Resource Construction Set (RCS) and saved to a .RSC file (see *Appendix C:* ***Native File Formats***) which is loaded by **rsrc_load()** at program initialization time.

Resources may also be embedded as data structures in source code (some utility programs convert .RSC files to source code). Desk accessories often do this to avoid complications they have in loading .RSC files.

Resources contain pointers and coordinates which must be fixed up before being used. **rsrc_load()** does this automatically, however if you use an embedded resource you must use **rsrc_rcfix()** if available or **rsrc_obfix()** on each object in each object tree to convert the initial character coordinates of to screen coordinates. This allows resources designed on screens with different aspect ratios and system fonts to appear the same. In any case, you should test your resources on several different screens, especially screen resolutions with different aspect ratios such as ST Medium and ST High.

Once a resource is loaded use **rsrc_gaddr()** to obtain pointers to individual object trees which can then be manipulated directly or with the **AES** Object Library. Replacing resources after they're loaded is accomplished with **rsrc_saddr()**.

Objects

Objects can be boxes, buttons, text, images, and more. An object tree is an array of **OBJECT** structures linked to form a structured relationship to each other. The **OBJECT** structure format is as follows:

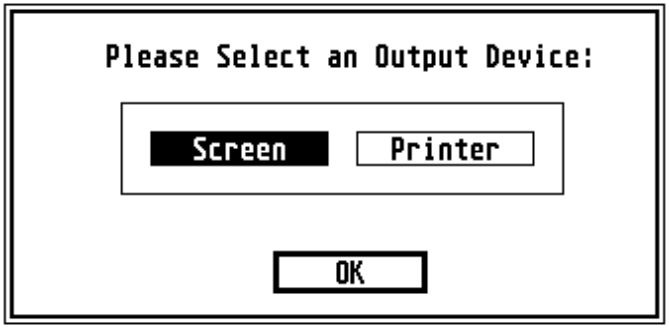
```
typedef struct object
{
       WORD      ob_next;
       WORD      ob_head;
       WORD      ob_tail;
       UWORD     ob_type;
       UWORD     ob_flags;
       UWORD     ob_state;
       VOIDP     ob_spec;
       WORD      ob_x;
       WORD      ob_y;
       WORD      ob_width;
       WORD      ob_height;
} OBJECT;
```

Normally **OBJECT**s are loaded in an application resource file but it is possible to create and manipulate them on-the-fly using the **objc_add()**, **objc_delete()**, and **objc_order()** commands.

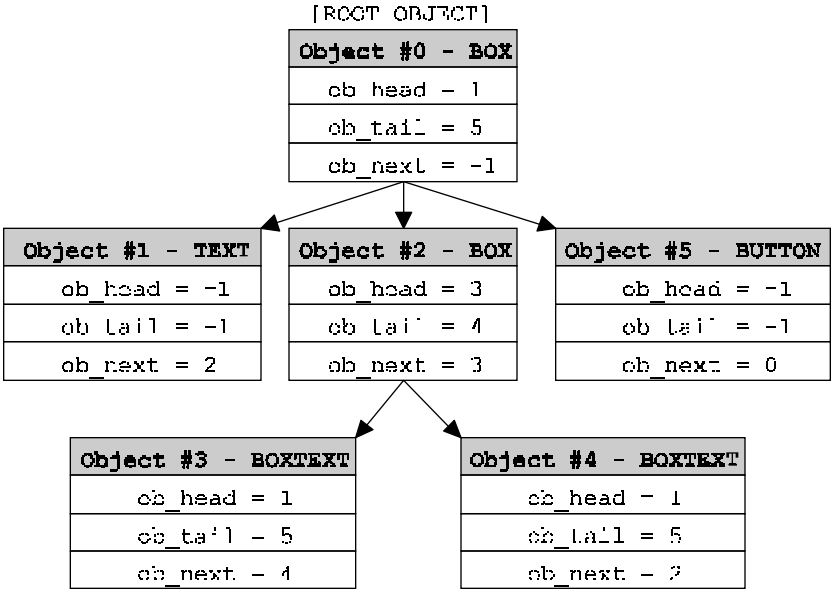
The first object in an **OBJECT** tree is called the **ROOT** object (**OBJECT** #0). It's coordinates are relative to the upper-left hand corner of the screen.

The **ROOT** object can have any number of children and each child can have children of their own. In each case, the **OBJECT**'s coordinates, *ob_x*, *ob_y*, *ob_width*, and *ob_height* are relative to that of its parent. The **AES** call **objc_offset()** can, however, be used to determine the exact screen coordinates of a child object. **objc_find()** is used to determine the object at a given screen coordinate.

The *ob_next*, *ob_head*, and *ob_tail* fields determine this relationship between parent **OBJECT**s and child **OBJECT**s. The following alert box is an example of an **OBJECT** tree:



The tree structure this object has can be represented as follows:



The exact usage of *ob_head*, *ob_next*, and *ob_tail* are as follows:

Element	Usage
ob_head	This member gives the exact index from the first object in the **OBJECT** tree to the first child of the current object. If the object has no children then this value should be -1.
ob_tail	This member gives the exact index from the first object in the **OBJECT** tree to the last child of the current object. If the object has no children then this value should be -1.
ob_next	This member gives the exact index from the first object in the **OBJECT** tree to the next child at the same level. The **ROOT** object should be set to -1. The last child at any given nesting level should be set to the index of its parent.

The low byte of the *ob_type* field specifies the object type as follows:

Name	*ob_type* & 0xFF	Meaning
G_BOX	20	Box
G_TEXT	21	Formatted Text
G_BOXTEXT	22	Formatted Text in a Box
G_IMAGE	23	Monochrome Image
G_PROGDEF	24	Programmer-Defined Object.

G_IBOX	25	Invisible Box
G_BUTTON	26	Push Button w/String
G_BOXCHAR	27	Character in a Box
G_STRING	28	Unformatted Text
G_FTEXT	29	Editable Formatted Text
G_FBOXTEXT	30	Editable Formatted Text in a Box
G_ICON	31	Monochrome Icon
G_TITLE	32	Menu Title
G_CICON	33	Color Icon (Available as of **AES** v3.3)

Object Flags

The *ob_flags* field of the **OBJECT** structure is a bitmask of different flags that can be applied to any object as follows:

Name	Bit(s)	Mask	Meaning
SELECTABLE	0	0x0001	Object's selected state may be toggled by clicking on it with the mouse.
DEFAULT	1	0x0002	An **EXIT** object with this bit set will have a thicker outline and be triggered when the user presses RETURN.
EXIT	2	0x0004	Clicking on this **OBJECT** and releasing the mouse button while still over it will cause the dialog to exit.
EDITABLE	3	0x0008	Set for **FTEXT** and **FBOXTEXT** objects to indicate that they may receive edit focus.
RBUTTON	4	0x0010	This object is one of a group of radio buttons. Clicking on it will deselect any selected objects at the same tree level that also have the **RBUTTON** flag set. Likewise, it will be deselected automatically when any other object is selected.
LASTOB	5	0x0020	This flag signals to the **AES** that the current **OBJECT** is the last in the object tree. (Required!)
TOUCHEXIT	6	0x0040	Setting this flag causes the **OBJECT** to return an exit state immediately after being clicked on with the mouse.
HIDETREE	7	0x0080	This **OBJECT** and all of its children will not be drawn.
INDIRECT	8	0x0100	This flag cause the *ob_spec* field to be interpreted as a pointer to the *ob_spec* value rather than the value itself.
FL3DIND	9	0x0200	Setting this flag causes the **OBJECT** to be drawn as a 3D indicator. This is appropriate for radio and toggle buttons. This flag is only recognized as of **AES** version 3.4.
FL3DACT	10	0x0400	Setting this flag causes the **OBJECT** to be drawn as a 3D activator. This is appropriate for **EXIT** buttons. This flag is only recognized as of **AES** version 3.4.

FL3DBAK	9 & 10	0x0600	If these bits are set, the object is treated as an **AES** background object. If it is **OUTLINED**, the outlined is drawn in a 3D manner. If its color is set to **WHITE** and its fill pattern is set to 0 then the **OBJECT** will inherit the default 3D background color. This flag is only recognized as of **AES** version 3.4.
SUBMENU	11	0x0800	This bit is set on menu items which have a sub-menu attachment. This bit also indicates that the high byte of the *ob_type* field is being used by the menu system.

Object States

The *ob_state* field determines the display state of the **OBJECT** as follows:

Name	Bit	Mask	Meaning
SELECTED	0	0x0001	The object is selected. An object with this bit set will be drawn in inverse video except for **G_CICON** which will use its 'selected' image.
CROSSED	1	0x0002	An **OBJECT** with this bit set will be drawn over with a white cross (this state can only usually be seen over a colored or **SELECTED** object).
CHECKED	2	0x0004	An **OBJECT** with this bit set will be displayed with a checkmark in its upper-left corner.
DISABLED	3	0x0008	An **OBJECT** with this bit set will ignore user input. Text objects with this bit set will draw in a dithered pattern.
OUTLINED	4	0x0010	**G_BOX**, **G_IBOX**, **G_BOXTEXT**, **G_FBOXTEXT**, and **G_BOXCHAR** **OBJECT**s with this bit set will be drawn with a double border.
SHADOWED	5	0x0020	**G_BOX**, **G_IBOX**, **G_BOXTEXT**, **G_FBOXTEXT**, and **G_BOXCHAR** **OBJECT**s will be drawn with a shadow.

The **AES** supports the **objc_change()** call which can be used to change the state of an object and (optionally) redraw it.

The Object-Specific Field

The *ob_spec* field contains different data depending on the object type as indicated in the table below:

Object	Contents of *ob_spec*
G_BOX	The low 16 bits contain a **WORD** containing color information for the **OBJECT**. Bits 23-16 contain a signed **BYTE** representing the border thickness of the box.
G_TEXT	The *ob_spec* field contains a pointer to a **TEDINFO** structure.
G_BOXTEXT	The *ob_spec* field contains a pointer to a **TEDINFO** structure.
G_IMAGE	The *ob_spec* field points to a **BITBLK** structure.
G_PROGDEF	The *ob_spec* field points to a **APPLBLK** structure.
G_IBOX	The low 16 bits contain a **WORD** containing color information for the **OBJECT**. Bits 23-16 contain a signed **BYTE** representing the border thickness of the box.
G_BUTTON	The *ob_spec* field contains a pointer to the text to be contained in the button.
G_BOXCHAR	The low 16 bits contain a **WORD** containing color information for the **OBJECT**. Bits 23-16 contain a signed **BYTE** representing the border thickness of the box. Bits 31-24 contain the ASCII value of the character to display.
G_STRING	The *ob_spec* field contains a pointer to the text to be displayed.
G_FTEXT	The *ob_spec* field contains a pointer to a **TEDINFO** structure.
G_FBOXTEXT	The *ob_spec* field contains a pointer to a **TEDINFO** structure.
G_ICON	The *ob_spec* field contains a pointer to an **ICONBLK** structure.
G_TITLE	The *ob_spec* field contains a pointer to the text to be used for the title.
G_CICON	The *ob_spec* field contains a pointer to a **CICONBLK** structure.

Object-Specific Structures

Almost all objects reference a **WORD** containing the object color as defined below (note the definition below may need to be altered depending upon the bit ordering of your compiler).

```
typedef struct objc_colorword
{
       UWORD     borderc  : 4;     /* Bits 15-12 contain the border color */
       UWORD     textc    : 4;     /* Bits 11-8 contain the text color */
       UWORD     opaque   : 1;     /* Bit 7 is 1 if opaque or 0 if transparent */
       UWORD     pattern  : 3;     /* Bits 6-4 contain the fill pattern index */
       UWORD     fillc    : 4;     /* Bits 3-0 contain the fill color */
} OBJC_COLORWORD;
```

Available colors for fill patterns, text, and borders are listed below:

Name	Value	Color
WHITE	0	White
BLACK	1	Black
RED	2	Red
GREEN	3	Green
BLUE	4	Blue
CYAN	5	Cyan
YELLOW	6	Yellow
MAGENTA	7	Magenta
LWHITE	8	Light Gray
LBLACK	9	Dark Gray
LRED	10	Light Red
LGREEN	11	Light Green
LBLUE	12	Light Blue
LCYAN	13	Light Cyan
LYELLOW	14	Light Yellow
LMAGENTA	15	Light Magenta

TEDINFO

G_TEXT, **G_BOXTEXT**, **G_FTEXT**, and **G_FBOXTEXT** objects all reference a **TEDINFO** structure in their *ob_spec* field. The **TEDINFO** structure is defined below:

```
typedef struct text_edinfo
{
       char *    te_ptext;
       char *    te_ptmplt;
       char *    te_pvalid;
       WORD      te_font;
       WORD      te_fontid;
       WORD      te_just;
       WORD      te_color;
       WORD      te_fontsize;
       WORD      te_thickness;
       WORD      te_txtlen;
       WORD      te_tmplen;
} TEDINFO;
```

The three character pointer point to text strings required for **G_FTEXT** and **G_FBOXTEXT** objects. *te_ptext* points to the actual text to be displayed and is the only field used by all text objects. *te_ptmplt* points to the text template for editable fields. For each character that the user can enter, the text string should contain a tilde character (ASCII 126). Other characters are displayed but cannot be overwritten by the user. *te_pvalid* contains validation characters for each character the user may enter. The current acceptable validation characters are:

Character	Allows
9	Digits 0-9
A	Uppercase letters A-Z plus SPACE
a	Upper and lowercase letters plus SPACE

N	Digits 0-9, uppercase letters A-Z, and SPACE
n	Digits 0-9, upper and lowercase letters A-Z, and SPACE
F	Valid **GEMDOS** filename characters plus question mark and asterisk
P	Valid **GEMDOS** pathname characters plus backslash, colon, question mark, and asterisk
p	Valid **GEMDOS** pathname characters plus backslash and colon
X	All characters

As an example the following diagram shows the correct text, template, and validation strings for obtaining a **GEMDOS** filename from the user.

String	Contents
te_ptext	'\0' (**NULL** char)
te_ptmplt	_____.___
te_pvalid	FFFFFFFFFFF

te_font may be set to any of the following values:

Name	*te_font*	Meaning
GDOS_PROP	0	Use a **SpeedoGDOS** font (valid only with an **AES** version of at least 4.0 and **SpeedoGDOS** installed).
GDOS_MONO	1	Use a **SpeedoGDOS** font (valid only with an **AES** version of at least 4.1 and **SpeedoGDOS** installed) and force monospaced output.
GDOS_BITM	2	Use a **GDOS** bitmap font (valid only with an **AES** version of at least 4.1 and **SpeedoGDOS** installed).
IBM	3	Use the standard monospaced system font.
SMALL	5	Use the small monospaced system font.

When using a value of **GDOS_PROP**, **GDOS_MONO**, or **GDOS_BITM**, *te_fontsize* specifies the font size in points and *te_fontid* specifies the **SpeedoGDOS** font identification number. Selecting the **IBM** or **SMALL** font will cause *te_fontsize* and *te_fontid* to be ignored.

te_just sets the justification of the text output as follows:

Name	*te_just*	Meaning
TE_LEFT	0	Left Justify
TE_RIGHT	1	Right Justify
TE_CNTR	2	Center

te_thickness sets the border thickness (positive and negative values are acceptable) of the **G_BOXTEXT** or **G_FBOXTEXT** object. *te_txtlen* and *te_tmplen* should be set to the length of the starting text and template length respectively.

BITBLK

G_IMAGE objects contain a pointer to a **BITBLK** structure in their *ob_spec* field. The **BITBLK** structure is defined as follows:

```
typedef struct bit_block
{
       WORD         *bi_pdata;
       WORD         bi_wb;
       WORD         bi_hl;
       WORD         bi_x;
       WORD         bi_y;
       WORD         bi_color;
} BITBLK;
```

bi_pdata should point to a monochrome bit image. *bi_wb* specifies the width (in bytes) of the image. All **BITBLK** images must be a multiple of 16 pixels wide therefore this value must be even.

bi_hl specifies the height of the image in scan lines (rows). *bi_x* and *bi_y* are used as offsets into *bi_pdata*. Any data occurring before these coordinates will be ignored. *bi_color* is a standard color **WORD** where the fill color specifies the color in which the image will be rendered.

ICONBLK

The *ob_spec* field of **G_ICON** objects point to an **ICONBLK** structure as defined below:

```
typedef struct icon_block
{
       WORD *      ib_pmask;
       WORD *      ib_pdata;
       char *      ib_ptext;
       WORD        ib_char;
       WORD        ib_xchar;
       WORD        ib_ychar;
       WORD        ib_xicon;
       WORD        ib_yicon;
       WORD        ib_wicon;
       WORD        ib_hicon;
       WORD        ib_xtext;
       WORD        ib_ytext;
       WORD        ib_wtext;
       WORD        ib_htext;
} ICONBLK;
```

ib_pmask and *ib_pdata* are pointers to the monochrome mask and image data respectively. *ib_ptext* is a string pointer to the icon text. *ib_char* defines the icon character (used for drive icons) and the icon foreground and background color as follows:

ib_char		
Bits 15-12	**Bits 11-8**	**Bits 7-0**
Icon Foreground Color	Icon Background Color	ASCII Character (or 0 for no character).

ib_xchar and *ib_ychar* specify the location of the icon character relative to *ib_xicon* and *ib_yicon*. *ib_xicon* and *ib_yicon* specify the location of the icon relative to the *ob_x* and *ob_y* of the object. *ib_wicon* and *ib_hicon* specify the width and height of the icon in pixels. As with images, icons must be a multiple of 16 pixels in width.

ib_xtext and *ib_ytext* specify the location of the text string relative to the *ob_x* and *ob_y* of the object. *ib_wtext* and *ib_htext* specify the width and height of the icon text area.

CICONBLK

The **G_CICON** object (available as of **AES** version 3.3) defines its *ob_spec* field to be a pointer to a **CICONBLK** structure as defined below:

```
typedef struct cicon_blk
{
       ICONBLK      monoblk;
       CICON *      mainlist;
} CICONBLK;
```

monoblk contains a monochrome icon which is rendered if a color icon matching the display parameters cannot be found. In addition, the icon text, character, size, and positioning data from the monochrome icon are always used for the color one. *mainlist* points to the first **CICON** structure in a linked list of color icons for different resolutions. **CICON** is defined as follows:

```
typedef struct cicon_data
{
       WORD                 num_planes;
       WORD *               col_data;
       WORD *               col_mask;
       WORD *               sel_data;
       WORD *               sel_mask;
       struct cicon_data *  next_res;
} CICON;
```

num_planes indicates the number of bit planes this color icon contains. *col_data* and *col_mask* point to the icon data and mask for the unselected icon respectively. Likewise, *sel_data* and *sel_mask* point to the icon data and mask for the selected icon. *next_res* points to the next color icon definition or **NULL** if no more are available. Bitmap data pointed to by these variables should be in **VDI** device-dependent format (they are stored as device-independent images in a .RSC file).

The **AES** searches the **CICONBLK** object for a color icon that has the same number of planes in the display. If none is found, the **AES** simply uses the monochrome icon.

APPLBLK

G_PROGDEF objects allow programmers to define custom objects and link them transparently in the resource. The *ob_spec* field of **G_PROGDEF** objects contains a pointer to an **APPLBLK** as defined below:

```
typedef struct appl_blk
{
       WORD        (*ab_code)(PARMBLK *);
       LONG        ab_parm;
} APPLBLK;
```

ab_code is a pointer to a user-defined routine which will draw the object. The routine will be passed a pointer to a **PARMBLK** structure containing the information it needs to render the object. The routine must be defined with stack checking off and expect to be passed its parameter on the stack. *ab_parm* is a user-defined value which is copied into the **PARMBLK** structure as defined below:

```
typedef struct parm_blk
{
       OBJECT      *tree;
       WORD        pb_obj;
       WORD        pb_prevstate;
       WORD        pb_currstate;
       WORD        pb_x;
       WORD        pb_y;
       WORD        pb_w;
       WORD        pb_h;
       WORD        pb_xc;
       WORD        pb_yc;
       WORD        pb_wc;
       WORD        pb_hc;
       LONG        pb_parm;
} PARMBLK;
```

tree points to the **OBJECT** tree of the object being drawn. The object is located at index *pb_obj*.

The routine is passed the old *ob_state* of the object in *pb_prevstate* and the new *ob_state* of the object in *pb_currstate*. If *pb_prevstate* and *pb_currstate* is equal then the object should be drawn completely, otherwise only the drawing necessary to redraw the object from *pb_prevstate* to *pb_currstate* are necessary.

pb_x, *pb_y*, *pb_w*, and *pb_h* give the screen coordinates of the object. *pb_xc*, *pb_yc*, *pb_wc*, and *pb_hc* give the rectangle to clip to. *pb_parm* contains a copy of the *ap_parm* value in the **APPLBLK** structure.

The custom routine should return a **WORD** containing any remaining *ob_state* bits you wish the **AES** to draw over your custom object.

Because the drawing routing will be called from the context of the **AES**, using the stack heavily or defining many local variables is not recommended.

Dialogs

Dialog boxes are modal forms of user input. This means that no other interaction can occur between the user and applications until the requirements of the dialog have been met and it is exited. A normal dialog box consists of an **OBJECT** tree with a **BOX** as its root object and any number of other controls that accept user input. Both alert boxes and the file selector are examples of **AES** provided dialog boxes.

The **AES form_do()** function is the simplest method of using a dialog box. Simply construct an **OBJECT** tree with at least one **EXIT** or **TOUCHEXIT** object and call **form_do()**. All interaction with the dialog like editable fields, radio buttons, and selectable objects will be maintained by the **AES** until the user strikes an **EXIT** or **TOUCHEXIT** object. The proper method for displaying a dialog box is shown in the example below:

```
WORD
do_dialog( OBJECT *tree, WORD first_edit )
{
        GRECT g;
        WORD ret;

        /* Reserve screen/mouse button */
        wind_update( BEG_UPDATE );
        wind_update( BEG_MCTRL );

        /* Center dialog on screen and put clipping rectangle in g */
        form_center( tree, &g.g_x, &g.g_y, &g.g_w, &g.g_h );

        /* Reserve screen space and draw growing box */
        form_dial( FMD_START, 0, 0, 0, 0, g.g_x, g.g_y, g.g_w, g.g_h );
        form_dial( FMD_GROW, g.g_x + g.g_w/2, g.g_y + g.g_h/2, 0, 0, g.g_x, g.g_y,
                  g.g_w, g.g_h );

        /* Draw the dialog box */
        objc_draw( tree, ROOT, MAX_DEPTH, g.g_x, g.g_y, g.g_w, g.g_h );

        /* Handle dialog */
        ret = form_do( tree, first_edit );

        /* Deselect EXIT button */
        tree[ret].ob_state &= ~SELECTED;

        /* Draw shrinking box and release screen area */
        form_dial( FMD_SHRINK, g.g_x + g.g_w/2, g.g_y + g.g_h/2, 0, 0, g.g_x, g.g_y,
                  g.g_w, g.g_h );
        form_dial( FMD_FINISH, 0, 0, 0, 0, g.g_x, g.g_y, g.g_w, g.g_h );

        /* Release screen/mouse control. */
        wind_update( END_MCTRL );
        wind_update( END_UPDATE );
```

```
        /* Return the object selected */
        return ret;
}
```

You may wish to create your own specialized dialog handling routines or place dialog boxes in windows to create modeless input. This can be accomplished by using the **form_button()**, **form_keybd()**, and **objc_edit() AES** calls. Specific information about these calls may be found in the ***Function Reference***.

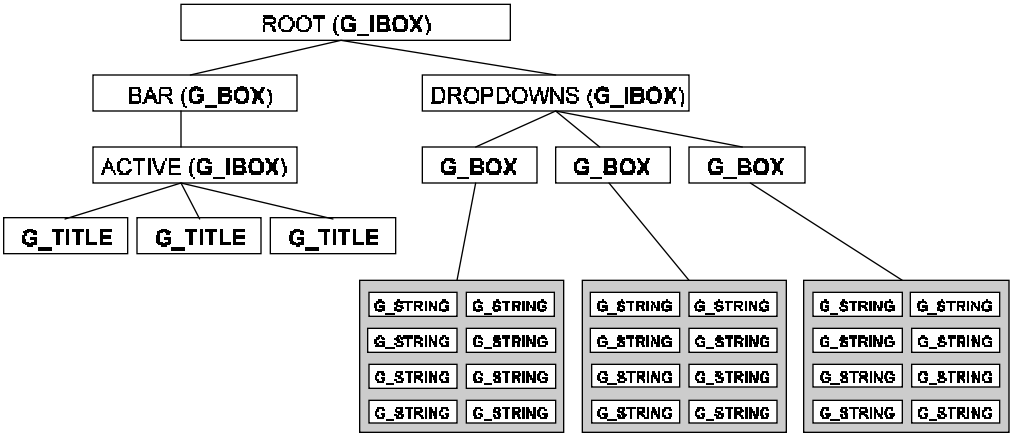
GEM also provides two generic dialog boxes through the **form_alert()** and **form_error()** calls. **form_alert()** displays an alert dialog with a choice between icons and user-defined text and buttons. **form_error()** displays an alert based on predefined system error codes.

Menus

Most **GEM** applications use a menu bar to allow the user to navigate through program options. In addition, newer versions of the **AES** now allow popup menus and drop-down list boxes (a special form of a popup menu). Menus are simply specially designed **OBJECT** trees activated using special **AES** calls.

The Menu Bar

The menu bar is a special **OBJECT** which is usually registered in the beginning stages of a **GEM** program which contains choices which the user may select to trigger a special menu event (**MN_SELECTED**) to be sent to the application's message loop. Normally, you will use a resource construction set to create a menu but if you are designing an **RCS** or must create a menu bar by hand, the format for the **OBJECT** structure of a **GEM** menu bar is shown below:



The **ROOT** object is a **G_IBOX** and should be set to the same width and height of the screen. It has two children, the **BAR** object and the **DROPDOWNS** object.

The **BAR** object is a **G_BOX** which should be the width of the screen and the height of the system font plus two pixels for a border line. The **DROPDOWNS** object is a **G_IBOX** and should be of a size large enough to encompass all of the drop-down menu boxes.

The **BAR** object has one child, the **ACTIVE** object, it should be the width of the screen and the height of the system font. It has as many **G_TITLE** children as there are menu titles.

The **DROPDOWNS** object has the same number of **G_BOX** child objects as the **ACTIVE** object has **G_TITLE** children. Each box must be high enough to support the number of **G_STRING** menu items and wide enough to support the longest item. Each **G_BOX** must be aligned so that it falls underneath its corresponding **G_TITLE**. In addition, each **G_STRING** menu item should be the same length as its parent **G_BOX** object.

Each **G_STRING** menu item should be preceded by two spaces. Each **G_TITLE** should be preceded and followed by one space. The first **G_BOX** object should appear under a **G_TITLE** object named 'Desk' and should contain eight children. The first child **G_STRING** is application defined (it usually leads to the 'About...' program credits), the second item should be a disabled separator ('----------') line. The next six items are dummy objects used by the **AES** to display program and desk accessory titles.

Utilizing a Menu Bar

Menu bars can be displayed and their handling initiated by calling **menu_bar()**. In addition, using this command, a menu bar may be turned off or replaced with another menu bar at any time.

Individual menu items may be altered with three **AES** calls. **menu_icheck()** sets or removes a checkmark from in front of menu items. **menu_ienable()** enables or disables a menu item. **menu_itext()** alters the text of a menu item. After receiving a message indicating that a menu item has been clicked, perform the action appropriate to the menu item and then call **menu_tnormal()** to return the menu title text to normal video.

Hierarchical Menus

AES versions 3.3 and above support hierarchical submenus. When a submenu is attached to a regular menu item, a right arrow is appended to the end of the menu item text and a submenu is displayed whenever the mouse is positioned over the menu item. The user may select submenu items which cause an extended version of the **MN_SELECTED** message to be delivered (containing the menu object tree).

Up to 64 submenu attachments may be in effect at any time per process. Attaching a single submenu to more than one menu item counts as only one attachment.

Submenus should be **G_BOX** objects with as many **G_STRING** (or other) child objects as necessary. One or several submenus may be contained in a single **OBJECT** tree. If the submenu's scroll flag is set, scroll arrows will appear and the menu will be scrollable if it

contains more items than the currently set system scroll value. Submenus containing user-defined objects should not have their scroll flag set.

Submenus are attached and removed with the **menu_attach()** call. A serious bug exists in **AES** versions lower than 4.0 which causes **menu_attach()** to crash the system if you use it to remove or inquire the state of an existing submenu. This means that submenus may only be removed in **AES** versions 4.0 and above. Submenus may be nested to up to four levels though only one level is recommended.

Submenus may not be attached to menu items in the left-most 'Desk' menu. Individual submenu items may be aligned with the parent object by using **menu_istart()**.

Popup Menus

AES versions 3.3 and above support popup menus. Popup menus share the same **OBJECT** structure as hierarchical menus but are never attached to a parent menu item. They may be displayed anywhere on the screen and are often called in response to selecting a special dialog item (see *Chapter 11: **GEM User Interface Guidelines***). Popup menus are displayed with the **AES** call **menu_popup()**.

Menu Settings

The **AES** call **menu_settings()** may be used to adjust certain global defaults regarding the appearance and timing delays of submenus and popup menus. Because this call affects all system applications it should only be utilized by a system configuration utility and not by individual applications.

Drop-Down List Boxes

AES versions 4.0 and later support a special type of popup menu called a drop-down list box. Setting the menu scroll flag to a value of -1 will cause a popup menu to be displayed as a drop-down list instead.

A drop-down list reveals up to eight items from a multiple item list to the user. A slider bar is displayed next to the list and is automatically handled during the **menu_popup()** call. Several considerations must be taken when using a drop-down list box:

- Drop-down lists may only contain **G_STRING** objects.
- If you want to force the AES to always draw scroll bars for the list box, the **OBJECT** tree must contain at least eight **G_STRING** objects. If less than that number of items exist, pad the remaining items with blanks and set the object's **DISABLED** flag.
- As long as the **OBJECT** tree has at least eight **G_STRING** objects, it should not be padded with any additional objects since the size of the slider is based on the number of objects.

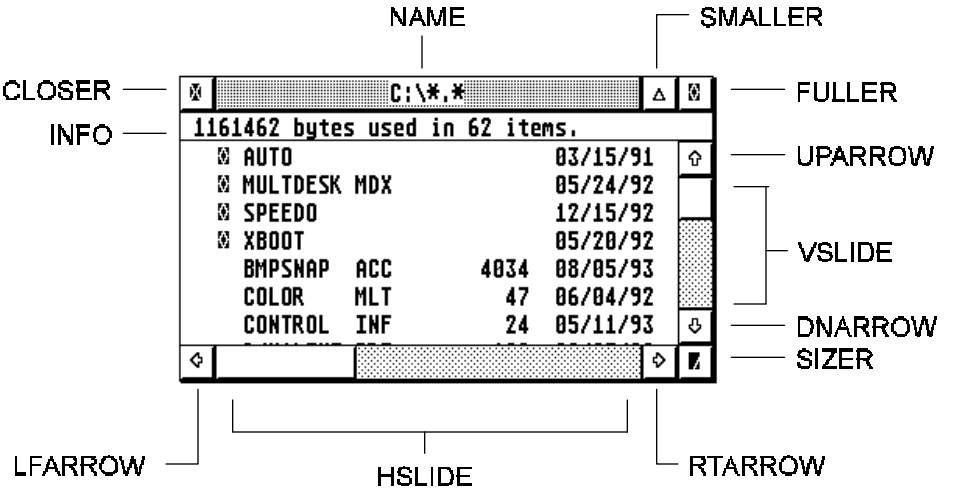
The Menu Buffer

A special memory area is allocated by the **AES** so that it may reserve the screen area underneath displayed menus. A pointer to this memory and its length may be obtained by calling **wind_get(WF_SCREEN**, ... **)**. Menu buffer memory may be used as a temporary holding arena for applications as long as the following rules are maintained:

- The application must not use a menu bar or it must be locked with **wind_update(BEG_UPDATE)**.
- Access to the menu buffer in a multitasking environment is not controlled so information stored by one application may be overwritten by another. It is therefore recommended that the menu buffer should not be used under **MultiTOS**.

Windows

GEM applications usually maintain most user-interaction in windows. Windows are workspaces created with **wind_create()** with any of several predefined gadgets (controls) illustrated in the diagram and table below:



Name	Mask	Meaning
NAME	0x0001	Using this mask will cause the **AES** to display the window with a title bar containing a name that the application should set with **wind_set(WF_NAME,** ... **)**.
CLOSER	0x0002	This mask will attach a closer box to the window which, when pressed, will send a **WM_CLOSED** message to the application.
FULLER	0x0004	This mask displays a fuller box with the window which, when pressed, will cause a **WM_FULLED** message to be sent to the application.
MOVER	0x0008	This mask allows the user to move the window by clicking and dragging on the window's title bar. This action will generate a **WM_MOVED** message.
INFO	0x0010	This mask creates an information line just below the title bar which can contain any user-defined information as set with **wind_set(WF_INFO**, ...**)**.
SIZER	0x0020	This mask attaches a sizer object to the window which, when clicked and dragged to a new location, will generate a **WM_SIZED** message.
UPARROW	0x0040	This mask attaches an up arrow object to the window which, when pressed, will generate a **WM_ARROWED** message to the application.
DNARROW	0x0080	This mask attaches a down arrow object to the window which, when pressed, will generate a **WM_ARROWED** message to the application.
VSLIDE	0x0100	This mask attaches a vertical slider object to the window which, when clicked and dragged, will generate a **WM_VSLID** message. Clicking on the exposed area of the slider will also generate this message.
LFARROW	0x0200	This mask attaches a left arrow object to the window which, when pressed, will generate a **WM_ARROWED** message to the application.
RTARROW	0x0400	This mask attaches a right arrow object to the window which, when pressed, will generate a **WM_ARROWED** message to the application.

HSLIDE	0x0800	This mask attaches a horizontal slider object to the window which, when clicked and dragged, will generate a **WM_HSLID** message. Clicking on the exposed area of the slider will also generate this message.
SMALLER	0x4000	This mask attaches a smaller object which, when clicked, will generate a **WM_ICONIFIED** message. If the object is CTRL-clicked, a **WM_ALLICONIFY** message will be generated. This object is only valid in **AES** v4.1 and higher.

wind_create() returns a window handle which should be stored as it must be referenced on any further calls that open, alter, close, or delete the window. **wind_create()** may fail if too many windows are already open. Different versions of the **AES** impose different limits on the number of concurrently open windows.

Calling **wind_create()** does not automatically display the window. **wind_open()** displays a window named by its window handle. Any calls needed to initialize the window (such as setting the window title, etc.) should be made between the **wind_create()** and **wind_open()** calls.

wind_set() and **wind_get()** can be used to set and retrieve many various window attributes. Look for their documentation in the function reference for further details.

wind_close() may be used to remove a window from the screen. The window itself and its attributes are not deleted as a result of this call, however. A subsequent call to **wind_open()** will restore a window to the state it was in prior to the **wind_close()** call. The **wind_delete()** function is used to physically delete a window and free any memory it was using.

Two other utility functions for use in dealing with windows are provided by the **AES**. **wind_calc()** will return the border rectangle of a window given the desired work area or the work area of a window given the desired border area. The call takes into account the sizes of the various window gadgets.

wind_find() returns the handle of the window currently under the mouse.

The Desktop Window

The desktop window encompasses the entire screen. It has a constant window handle of **DESK** (0) so information about it can be inquired with **wind_get()**. Calling **wind_get()** with a parameter of **WF_CURRXYWH** will return the size of the screen. Calling **wind_get()** with a parameter of **WF_WORKXYWH** will return the size of the screen minus the size of the menu bar.

The desktop draws a custom **OBJECT** tree in its work area. This tree results in the fill pattern and color seen on screen. An application may create its own custom desktop object tree by using **wind_set()** with a parameter of **WF_DESKTOP**. The **OBJECT** tree specified should be the exact size of the desktop work area.

MultiTOS will switch between these object trees as applications are switched. The desktop's object tree will be visible whenever an application doesn't specify one of its own.

The Rectangle List

Whenever a window receives a redraw message or needs to update its window because of its reasons, it should always constrain output to its current rectangle list. The **AES** will calculate the size and position of a group of rectangles that compromise the area of your window not covered by other overlapping windows.

wind_get() with parameters of **WF_FIRSTXYWH** and **WF_NEXTXYWH** is used to return the current rectangle list. Redrawing inside a window should also only be attempted when the window semaphore is locked with **wind_update(BEG_UPDATE)**. This prevents the rectangle list from changing during the redraw and prevents the user from dropping down menus which might be overwritten. The following code sample illustrates a routine that correctly steps through the rectangle list:

```
.
.
. Application Event Loop
.
case WM_REDRAW:
      RedrawWindow( msg[3], (GRECT *)&msg[4] );
      break;
.
.
```

```
VOID
RedrawWindow( WORD winhandle, GRECT *dirty )
{
       GRECT rect;

       wind_update( BEG_UPDATE );

       wind_get( winhandle, WF_FIRSTXYWH, &rect.g_x, &rect.g_y, &rect.g_w,
       &rect.g_h);
       while( rect.g_w && rect.g_h )
       {
```

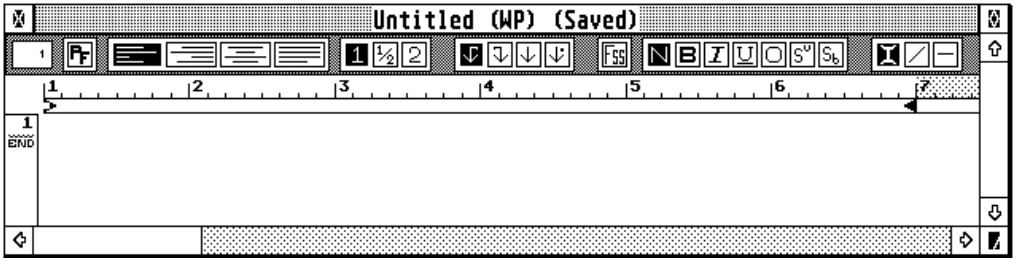
```
            if( rc_intersect( dirty, &rect ) )
            {
                /*
                 * Do your drawing here...constrained to the rectangle in g.
                 */
            }

            wind_get( winhandle, WF_NEXTXYWH, &rect.g_x, &rect.g_y, &rect.g_w,
                    &rect.g_h);
        }

        wind_update( END_UPDATE );
}
```

Window Toolbars

AES versions 4.0 and later support window toolbar attachments. Toolbars are **OBJECT** trees containing a number of **TOUCHEXIT** objects. They are attached to a window using **wind_set()** with a parameter of **WF_TOOLBAR**. The following diagram shows a window with a toolbar:



Example from Atari Works 2.1

Window toolbars are automatically redrawn whenever necessary and their **ROOT** objects are automatically repositioned and resized with the window. If any special redrawing is necessary (ex: changing the visual state of an object after a click), the application may obtain a special toolbar rectangle list by using **wind_get()** with parameters of **WF_FTOOLBAR** and **WF_NTOOLBAR**.

If toolbar objects must be modified on **WM_SIZED** events, simply modify them prior to calling **wind_set(** handle, **WM_CURRXYWH**, ... **)**.

A special note about windows with toolbars concerns the usage of **wind_calc()**. **wind_calc()** doesn't understand the concept of toolbars. The information it returns must be modified by adjusting the height of its output rectangles according to the current height of the toolbar object tree.

The Graphics Library

The Graphics Library contain many functions which can be used to provide visual clues to the user. This library also contains functions to inquire and set information about the mouse pointer.

graf_movebox(), **graf_shrinkbox()**, and **graf_growbox()** display animations that can be used to indicate an impending change in the screen display. **graf_dragbox()**, **graf_rubberbox()**, and **graf_slidebox()** display visual effects that are interactively changed by the mouse position.

graf_mkstate() is used to inquire the current state of the mouse buttons and mouse position. **graf_mouse()** can be used to change the shape of the system mouse. **graf_handle()** is used to return the physical handle of the screen (needed to open a **VDI** workstation) and the metrics of the system default text font.

The File Selector Library

Two routines are provided by the **AES** to display and handle the common system file selector. **AES** versions less than 1.4 do not support **fsel_exinput()**. All **AES** versions support **fsel_input()**.

Both calls take a **GEMDOS** pathname and filename as parameters. The pathname should include a complete path specification including a drive letter, colon, path, and filemask. The filemask may (and usually does include wildcard characters). The application may also pass a default filename to the selector.

fsel_exinput() allows the application to specify a replacement title for the file selector which reminds the user about the action they are taking such as 'Select a .DOC file to open...'.

The Scrap Library

The **scrp_read()** and **scrp_write()** calls are provided by the **AES** to return and set the current clipboard path. The clipboard is a global resource in which applications can share data. Applications supporting the clipboard contain an 'Edit' menu title which has at least the following four items, 'Cut', 'Copy', 'Paste', and 'Delete'. An appropriate action for each is listed below:

Implementing 'Cut' and 'Copy'

When the user selects 'Cut' or 'Copy' from the 'Edit' menu and an object is selected ('Cut' and 'Copy' should only be enabled in the menu when an object is selected which may be transferred to the clipboard) the following steps may be used to transfer the data to the system clipboard:

1. Call **scrp_read()** to return the name of the current scrap directory. If the returned string is empty, no clipboard directory has been defined since the computer has

been started. The directory string returned may need to be reformatted. A proper directory string ends in a backslash, however some applications incorrectly append a filename to this string.

2. If no clipboard directory was returned or the one specified is invalid, create a directory in the user's boot drive called '\CLIPBRD' and write the pathname back using **scrp_write()**. For example, if the user's boot drive was 'C:' then your parameter to **scrp_write()** would be 'C:\CLIPBRD\'.
3. Search and delete files in the current clipboard directory with the mask 'SCRAP.*'.
4. Now write a disk file for the selected data to a file named SCRAP.??? where '???' is the proper file extension for an object of its type. If the object can be represented in more than one format by your application, write as many formats as possible all named 'SCRAP' with the proper file extension.
5. If the menu choice was 'Cut' rather than 'Copy,' delete the object from your data structures and update your application as necessary.

Implementing 'Paste'

'Paste' is used to read a file and insert it appropriately into an application that supports data of its type. To implement 'Paste' follow the steps below:

1. Call **scrp_read()** to obtain the current system clipboard directory. If the returned string is empty, no data is in the clipboard.
2. Format the string returned by **scrp_read()** into a usable pathname and search for files called 'SCRAP' in that path having a file extension of data that your application supports. Remember, more than one SCRAP.??? file may be present.
3. Load the data and insert it in your application as appropriate.

MultiTOS Notes

The **AES**, when running under **MultiTOS**, will create a **MiNT** semaphore named '_SCP' which should be used to provide negotiated access to the scrap directory. Access to this semaphore should be obtained from MiNT prior to any clipboard operation and must be released as soon as it is complete. Applications should not attempt to destroy this semaphore.

The Shell Library

The Shell Library was originally intended to provide **AES** support to the Desktop application. Many of the routines, however, are useful to other **GEM** applications. Some functionality of the Shell Library was discussed earlier in this chapter in 'The Environment String'.

The Shell Buffer

The Desktop application loads the DESKTOP.INF or NEWDESK.INF file (depending on the **TOS** version) into the shell buffer. Prior to **TOS** 2.00, the shell buffer was 1024 bytes long meaning that was the maximum length of the DESKTOP.INF file. **AES** versions 2.00 to 3.30 allocate a buffer 4096 bytes long. **AES** versions 3.30 and above support variable-length buffers.

The shell buffer contains the 'working' copy of the above mentioned system files. The information in this buffer may be copied by using **shel_get()**. Likewise, information can be written to this buffer using **shel_put()**. Extreme care must be used with these functions as their misuse can confuse or possibly even crash the Desktop.

Miscellaneous Shell Library Functions

shel_find() is used to locate data files associated with an application. The **AES** uses this call to locate application resource files during **rsrc_load()**.

shel_read() returns information about the process which called the application (usually the Desktop).

shel_write() was originally used only to spawn new applications. With newer **AES** versions, though, **shel_write()** has taken on an enormous functionality and its documentation should be consulted for more information.

The GEM.CNF File

When running under **MultiTOS**, the **AES** will load and process an ASCII text file called 'GEM.CNF' which contains command lines that set environment and **AES** system variables and may run **GEM** programs. In addition, a replacement shell program may be specified in this file (see *Chapter 9: **Desktop*** for more information).

AES environment variables may be set in the 'GEM.CNF' file with the command 'setenv' as in the following example:

```
setenv TOSRUN=c:\multitos\miniwin.app
```

Several **AES** system variables may also be set in this file as shown in the following example:

```
AE_FONTID=3
```

Currently recognized **AES** system variables that may be set are shown in the following table:

Variable	Meaning
AE_FONTID	This variable may be set to any valid Speedo outline font ID which will be used as the **AES** default text font. This feature is only valid as of **AES** version 4.1.
AE_PNTSIZE	This variable defines the size of the **AES** default text font in points. This feature is only valid as of **AES** version 4.1.
AE_SREDRAW	Setting this variable to 1 causes the **AES** to send a full-screen redraw message whenever an application starts. Setting it to 0 disables this feature. The default is 1.
AE_TREDRAW	Setting this variable to 1 causes the **AES** to send a full-screen redraw message whenever an application terminates. Setting it to 0 disables this feature. The default is 1.

The 'GEM.CNF' file may also be used to automatically start applications as shown in the following example:

```
run c:\multitos\tclock.prg
```

AES Function Calling Procedure

The **GEM AES** is accessed through a 680x0 TRAP #2 statement. Upon calling the TRAP, register d0 should contain the magic number 0xC8 and register d1 should contain a pointer to the **AES** parameter block. The *global* data array member of the parameter block is filled in with information about the **AES** after an **appl_init()** call (see **appl_init()** for more details). The **AES** parameter block is a structure containing pointers to several arrays defined as follows:

```
struct aespb
{
        WORD *contrl;
        WORD *global;
        WORD *intin;
        WORD *intout;
        LONG *addrin;
        LONG *addrout;
};
```

The *control* array is filled in prior to an **AES** call with information about the number of parameters the function is being passed, the number of return values the function expects, and the opcode of the function itself as follows:

contrl[x]	Contents
0	Function opcode.
1	Number of *intin* elements the function is being sent.
2	Number of *intout* elements the function is being sent.
3	Number of *addrin* elements the function returns.
4	Number of *addrout* elements the function returns.

The *intin* array and *addrin* arrays are used to pass integer and address parameters respectively (consult each individual binding for details).

Upon return from the call, the *intout* and *addrout* arrays will be filled in with any appropriate output values.

To add a binding for the **AES** to your compiler you will usually write a short procedure that provides an interface to the **AES** arrays. The following example illustrates the binding to **graf_dragbox()** in this manner:

```
WORD
graf_dragbox( WORD width, WORD height, WORD start_x, WORD start_y,
                WORD box_x, WORD box_y, WORD box_w, WORD box_h,
                WORD *end_x, WORD *end_y )
{
        contrl[0] = 71;
        contrl[1] = 8;
        contrl[2] = 3;
        contrl[3] = 0;
        contrl[4] = 0;

        intin[0] = width;
        intin[1] = height;
        intin[2] = start_x;
        intin[3] = start_y;
        intin[4] = box_x;
        intin[5] = box_y;
        intin[6] = box_w;
        intin[7] = box_h;

        aes();

        *end_x = intout[1];
        *end_y = intout[2];

        return intout[0];
}
```

The following code is the assembly language function **aes()** used by the function above:

```
              .globl       _aes

              .text
_aes:
              lea          _aespb,a0
              move.l       a0,d1
              move.w       #$C8,d0
              trap         #2
              lea          _intout,a0
              move.w       (a0),d0
              rts

              .data

_aespb:       .dc.l        _contrl, _global, _intin, _intout, _addrin, _addrout

              .bss

_contrl:      .ds.w        5
_global:      .ds.w        15
_intin:       .ds.w        16
_intout:      .ds.w        7
_addrin:      .ds.l        2
_addrout:     .ds.l        1

              .end
```

The bindings in the ***AES Function Reference*** call a specialized function called **crys_if()** to actually call the **AES**. Many compilers use this method as well (Lattice C calls the function **_AESif()**).

crys_if() properly fills in the *contrl* array and calls the **AES**. It is passed one **WORD** parameter in d0 which contains the opcode of the function minus ten multiplied by four (for quicker table indexing). This gives an index into a table from which the *contrl* array data may be loaded. The **crys_if()** function is listed below:

```
* Note that this binding depends on the fact that no current AES call utilizes
* the addrout array

              .globl   _crys_if
              .globl   _aespb
              .globl   _contrl
              .globl   _global
              .globl   _intin
              .globl   _addrin
              .globl   _intout
              .globl   _addrout

              .text

_crys_if:
              lea      table(pc),a0     ; Table below
```

```
            move.l   0(a0,d0.w),d0        ; Load four packed bytes into d0
            lea      _aespb,a0          ; Load address of _aespb into a0
            movea.l  (a0),a1            ; Move address of contrl into a1
            movep.l  d0,1(a1)           ; Move four bytes into WORDs at 1(contrl)
            move.l   a0,d1              ; Move address of _aespb into d1
            move.w   #$C8,d0            ; AES magic number
            trap     #2                 ; Call GEM
            lea      _intout,a0         ; Get return value
            move.w   (a0),d0            ; Put it into d0
            rts

* Table of AES opcode/control values
* Values are: opcode, intin, intout, addrin
* As stated before, addrout is left at 0 since no AES calls use it

table:
            .dc.b       10, 0, 1, 0     ; appl_init
            .dc.b       11, 2, 1, 1     ; appl_read
            .dc.b       12, 2, 1, 1     ; appl_write
            .dc.b       13, 0, 1, 1     ; appl_find
            .dc.b       14, 2, 1, 1     ; appl_tplay
            .dc.b       15, 1, 1, 1     ; appl_trecord
            .dc.b       16, 0, 0, 0     ;
            .dc.b       17, 0, 0, 0     ;
            .dc.b       18, 1, 3, 1     ; appl_search (v4.0)
            .dc.b       19, 0, 1, 0     ; appl_exit
            .dc.b       20, 0, 1, 0     ; evnt_keybd
            .dc.b       21, 3, 5, 0     ; evnt_button
            .dc.b       22, 5, 5, 0     ; evnt_mouse
            .dc.b       23, 0, 1, 1     ; evnt_mesag
            .dc.b       24, 2, 1, 0     ; evnt_timer
            .dc.b       25, 16, 7, 1    ; evnt_multi
            .dc.b       26, 2, 1, 0     ; evnt_dclick
            .dc.b       27, 0, 0, 0     ;
            .dc.b       28, 0, 0, 0     ;
            .dc.b       29, 0, 0, 0     ;
            .dc.b       30, 1, 1, 1     ; menu_bar
            .dc.b       31, 2, 1, 1     ; menu_icheck
            .dc.b       32, 2, 1, 1     ; menu_ienable
            .dc.b       33, 2, 1, 1     ; menu_tnormal
            .dc.b       34, 1, 1, 2     ; menu_text
            .dc.b       35, 1, 1, 1     ; menu_register
            .dc.b       36, 2, 1, 2     ; menu_popup (v3.3)
            .dc.b       37, 2, 1, 2     ; menu_attach (v3.3)
            .dc.b       38, 3, 1, 1     ; menu_istart (v3.3)
            .dc.b       39, 1, 1, 1     ; menu_settings (v3.3)
            .dc.b       40, 2, 1, 1     ; objc_add
            .dc.b       41, 1, 1, 1     ; objc_delete
            .dc.b       42, 6, 1, 1     ; objc_draw
            .dc.b       43, 4, 1, 1     ; objc_find
            .dc.b       44, 1, 3, 1     ; objc_offset
            .dc.b       45, 2, 1, 1     ; objc_order
            .dc.b       46, 4, 2, 1     ; objc_edit
            .dc.b       47, 8, 1, 1     ; objc_change
            .dc.b       48, 4, 3, 0     ; objc_sysvar (v3.4)
            .dc.b       49, 0, 0, 0     ;
            .dc.b       50, 1, 1, 1     ; form_do
            .dc.b       51, 9, 1, 0     ; form_dial
            .dc.b       52, 1, 1, 1     ; form_alert
```

```
.dc.b       53, 1, 1, 0      ; form_error
.dc.b       54, 0, 5, 1      ; form_center
.dc.b       55, 3, 3, 1      ; form_keybd
.dc.b       56, 2, 2, 1      ; form_button
.dc.b       57, 0, 0, 0      ;
.dc.b       58, 0, 0, 0      ;
.dc.b       59, 0, 0, 0      ;
.dc.b       60, 0, 0, 0      ;
.dc.b       61, 0, 0, 0      ;
.dc.b       62, 0, 0, 0      ;
.dc.b       63, 0, 0, 0      ;
.dc.b       64, 0, 0, 0      ;
.dc.b       65, 0, 0, 0      ;
.dc.b       66, 0, 0, 0      ;
.dc.b       67, 0, 0, 0      ;
.dc.b       68, 0, 0, 0      ;
.dc.b       69, 0, 0, 0      ;
.dc.b       70, 4, 3, 0      ; graf_rubberbox
.dc.b       71, 8, 3, 0      ; graf_dragbox
.dc.b       72, 6, 1, 0      ; graf_movebox
.dc.b       73, 8, 1, 0      ; graf_growbox
.dc.b       74, 8, 1, 0      ; graf_shrinkbox
.dc.b       75, 4, 1, 1      ; graf_watchbox
.dc.b       76, 3, 1, 1      ; graf_slidebox
.dc.b       77, 0, 5, 0      ; graf_handle
.dc.b       78, 1, 1, 1      ; graf_mouse
.dc.b       79, 0, 5, 0      ; graf_mkstate
.dc.b       80, 0, 1, 1      ; scrp_read
.dc.b       81, 0, 1, 1      ; scrp_write
.dc.b       82, 0, 0, 0      ;
.dc.b       83, 0, 0, 0      ;
.dc.b       84, 0, 0, 0      ;
.dc.b       85, 0, 0, 0      ;
.dc.b       86, 0, 0, 0      ;
.dc.b       87, 0, 0, 0      ;
.dc.b       88, 0, 0, 0      ;
.dc.b       89, 0, 0, 0      ;
.dc.b       90, 0, 2, 2      ; fsel_input
.dc.b       91, 0, 2, 3      ; fsel_exinput
.dc.b       92, 0, 0, 0      ;
.dc.b       93, 0, 0, 0      ;
.dc.b       94, 0, 0, 0      ;
.dc.b       95, 0, 0, 0      ;
.dc.b       96, 0, 0, 0      ;
.dc.b       97, 0, 0, 0      ;
.dc.b       98, 0, 0, 0      ;
.dc.b       99, 0, 0, 0      ;
.dc.b       100, 5, 1, 0     ; wind_create
.dc.b       101, 5, 1, 0     ; wind_open
.dc.b       102, 1, 1, 0     ; wind_close
.dc.b       103, 1, 1, 0     ; wind_delete
.dc.b       104, 2, 5, 0     ; wind_get
.dc.b       105, 6, 1, 0     ; wind_set
.dc.b       106, 2, 1, 0     ; wind_find
.dc.b       107, 1, 1, 0     ; wind_update
.dc.b       108, 6, 5, 0     ; wind_calc
.dc.b       109, 0, 0, 0     ; wind_new
.dc.b       110, 0, 1, 1     ; rsrc_load
.dc.b       111, 0, 1, 0     ; rsrc_free
```

```
            .dc.b       112, 2, 1, 0     ; rsrc_gaddr
            .dc.b       113, 2, 1, 1     ; rsrc_saddr
            .dc.b       114, 1, 1, 1     ; rsrc_obfix
            .dc.b       115, 0, 0, 0     ; rsrc_rcfix (v4.0)
            .dc.b       116, 0, 0, 0     ;
            .dc.b       117, 0, 0, 0     ;
            .dc.b       118, 0, 0, 0     ;
            .dc.b       119, 0, 0, 0     ;
            .dc.b       120, 0, 1, 2     ; shel_read
            .dc.b       121, 3, 1, 2     ; shel_write
            .dc.b       122, 1, 1, 1     ; shel_get
            .dc.b       123, 1, 1, 1     ; shel_put
            .dc.b       124, 0, 1, 1     ; shel_find
            .dc.b       125, 0, 1, 2     ; shel_envrn
            .dc.b       126, 0, 0, 0     ;
            .dc.b       127, 0, 0, 0     ;
            .dc.b       128, 0, 0, 0     ;
            .dc.b       129, 0, 0, 0     ;
            .dc.b       130, 1, 5, 0     ; appl_getinfo (v4.0)

            .data

_aespb:     .dc.l       _contrl, _global, _intin, _intout, _addrin, _addrout
_contrl:    .dc.l       0, 0, 0, 0, 0

            .bss

* _contrl = opcode
* _contrl+2 = num_intin
* _contrl+4 = num_addrin
* _contrl+6 = num_intout
* _contrl+8 = num_addrout

_global     .ds.w       15
_intin      .ds.w       16
_intout     .ds.w       7
_addrin     .ds.l       2
_addrout    .ds.l       1

            .end
```