AES Function Reference

Application Services Library

The *Application Services Library* provides general use functions used in locating and working with other resident applications in addition to providing **AES** initialization and termination code. The members of the *Application Services Library* are:

- **appl_exit()**
- **appl_find()**
- **appl_getinfo()**
- **appl_init()**
- **appl_read()**
- **appl_search()**
- **appl_tplay()**
- **appl_trecord()**
- **appl_write()**

appl_exit()

WORD appl_exit(VOID)

appl_exit() should be called at the termination of any program initialized with **appl_init()**.

OPCODE 19 (0x13)

AVAILABILITY All **AES** versions.

BINDING

```
return crys_if(0x13);
```

RETURN VALUE **appl_exit()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS The proper procedure for handling an error from this function is currently undefined.

SEE ALSO **appl_init()**

appl_find()

WORD appl_find(*fname*)
CHAR *_fname_;

appl_find() searches the **AES**'s current process list for a program named *fname* and, if present, returns the application identifier of the process.

OPCODE 13 (0x0D)

AVAILABILITY All **AES** versions.

PARAMETERS *fname* is a pointer to a null-terminated ASCII string containing a valid **GEMDOS** filename (not including an extension) padded with blanks to be exactly 8 characters long (not including the **NULL**).

BINDING

```
addrin[0] = fname;

return crys_if(0x0D);
```

RETURN VALUE **appl_find()** returns the application identifier of the process if it is found or -1 otherwise.

VERSION NOTES **AES** versions from 4.0 add several extensions to this call for the benefit of **MultiTOS** as follows:

- If the upper word of the **CHAR** * is 0xFFFF, the lower word is assumed to be the **MiNT** id and **appl_find()** will return the **AES** application identifier.
- If the upper word of the **CHAR** * is 0xFFFE, the lower word is assumed to be the **AES** application identifier and the **MiNT** id is returned.
- If the upper word of the **CHAR** * is 0x0000, the current processes' application identifier is returned.

This functionality only exists if the **AES** version is 4.0 and above and **appl_getinfo()** indicates that it is available.

SEE ALSO **appl_write()**, **appl_init()**

appl_getinfo()

WORD appl_getinfo(*ap_gtype, ap_gout1, ap_gout2, ap_gout3, ap_gout4*)
WORD *ap_gtype*;
WORD **ap_gout1, *ap_gout2, *ap_gout3, *ap_gout4*;

appl_getinfo() returns information about the **AES**.

OPCODE 130 (0x82)

AVAILABILITY Available as of **AES** version 4.00.

PARAMETERS *ap_gtype* specifies the type of information to be returned in the shorts pointed to by *ap_gout1*, *ap_gout2*, *ap_gout3*, and *ap_gout4* as follows:

Name	Value	Returns
AES_LARGEFONT	0	**AES Large Font Information** *ap_gout1* is filled in with the **AES** font's point size. *ap_gout2* is filled in with the font id. *ap_gout3* is a code indicating the type of font: **SYSTEM_FONT** (0) is the system font **OUTLINE_FONT** (1) is an outline font *ap_gout4* is unused.
AES_SMALLFONT	1	**AES Large Font Information** Same as above for the current small font.

AES_SYSTEM	2	**AES System Specifics** *ap_gout1* is filled in with the resolution number (as would be returned by **Getrez()).** *ap_gout2* is filled in with the number of colors supported by the **AES** object library. *ap_gout3* is 0 if color icons are not supported or 1 if they are. *ap_gout4* is 0 to indicate that the extended resource file format is not supported or 1 if it is.
AES_LANGUAGE	3	**AES Globalization** *ap_gout1* is filled in with the current **AES** language code as follows: **Name** — ***ap_gout1*** — **Language** **AESLANG_ENGLISH** — 0 — English **AESLANG_GERMAN** — 1 — German **AESLANG_FRENCH** — 2 — French — — 3 — (Reserved) **AESLANG_SPANISH** — 4 — Spanish **AESLANG_ITALIAN** — 5 — Italian **AESLANG_SWEDISH** — 6 — Swedish *ap_gout2*, *ap_gout3*, and *ap_gout4* are unused.
AES_PROCESS	4	**AES Multiple Process Support** *ap_gout1* is 0 to indicate the use of non-pre-emptive multitasking and 1 to indicate the use of pre-emptive multitasking. *ap_gout2* is 0 if **appl_find()** cannot convert between **MiNT** and **AES** id's and 1 to indicate that it can. *ap_gout3* is 0 if **appl_search()** is not implemented and 1 if it is. *ap_gout4* is 0 if **rsrc_rcfix()** is not implemented and 1 if it is.
AES_PCGEM	5	**AES PC-GEM Features** *ap_gout1* is 0 if **objc_xfind()** is not implemented and 1 if it is. *ap_gout2* is currently reserved. *ap_gout3* is 0 if **menu_click()** is not implemented and 1 if it is. *ap_gout4* is 0 if **shel_rdef()** and **shel_wdef()** are not implemented and 1 if they are.

AES_INQUIRE	6	**AES Extended Inquiry Functions** *ap_gout1* is 0 if -1 is not a valid *ap_id* parameter to **appl_read()** or 1 if it is. *ap_gout2* is 0 if -1 is not a valid length parameter to **shel_get()** or 1 if it is. *ap_gout3* is 0 if -1 is not a valid *mode* parameter to **menu_bar()** or 1 if it is. *ap_gout4* is 0 if **MENU_INSTL** is not a valid *mode* parameter to **menu_bar()** or 1 if it is.
–	7	Currently reserved.
AES_MOUSE	8	**AES Mouse Support** *ap_gout1* is 0 to indicate that *mode* parameters of 258-260 are not supported by **graf_mouse()** and 1 if they are. *ap_gout2* is 0 to indicate that the application has control over the mouse form and 1 to indicate that the mouse form is maintained by the **AES** on a per-application basis. *ap_gout3* and *ap_gout4* are currently unused.
AES_MENU	9	**AES Menu Support** *ap_gout1* is 0 to indicate that sub-menus are not supported and 1 if **MultiTOS** style sub-menus are. *ap_gout2* is 0 to indicate that popup menus are not supported and 1 if **MultiTOS** style popup menus are. *ap_gout3* is 0 to indicate that scrollable menus are not supported and 1 if **MultiTOS** style scrollable menus are. *ap_gout4* is 0 to indicate that the **MN_SELECTED** message does not contain object tree information in *msg*[5-7] and 1 to indicate that it does.

AES_SHELL	10	**AES Shell Support** *ap_gout1* & 0x00FF indicates the highest legal value for the *mode* parameter of **shel_write()**. *ap_gout1* & 0xFF00 indicate which extended **shel_write()** *mode* bits are supported. *ap_gout2* is 0 if **shel_write()** with a *mode* parameter of 0 launches an application or 1 if it cancels the previous **shel_write()**. *ap_gout3* is 0 if **shel_write()** with a *mode* parameter of 1 launches an application immediately or 1 if it takes effect when the current application exits. *ap_gout4* is 0 if ARGV style parameter passing is not supported or 1 if it is.
AES_WINDOW	11	**AES Window Features** *ap_gout1* is a bitmap of extended modes supported by **wind_get()** and **wind_set()** (if a bit is set, it is supported) as follows: **Bit** — ***mode*** 0 — **WF_TOP** returns window below the top also. 1 — **wind_get(WF_NEWDESK**, ... **)** supported. 2 — **WF_COLOR** get/set. 3 — **WF_DCOLOR** get/set. 4 — **WF_OWNER** get/set. 5 — **WF_BEVENT** get/set. 6 — **WF_BOTTOM** set. 7 — **WF_ICONIFY** set. 8 — **WF_UNICONIFY** set. 9-15 — Unused *ap_gout2* is current unused. *ap_gout3* is a bitmap of supported window behaviors (if a bit is set, it is supported) as follows: **Bit** — **Behaviour** 0 — Iconifier gadget present. 1 — Bottomer gadget present. 2 — SHIFT-click sends window to bottom. 3 — “hot” close box supported. 4-15 — Unused *ap_gout4* is currently unused.

AES_MESSAGE	12	**AES Extended Messages** *ap_gout1* is a bitmap of extra messages supported (if a bit is set, it is supported) as follows: **Bit** **Message** 0 **WM_NEWTOP** is meaningful. 1 **WM_UNTOPPED** is sent. 2 **WM_ONTOP** is sent. 3 **AP_TERM** is sent. 4 Shutdown and resolution change messages. 5 **CH_EXIT** is sent. 6 **WM_BOTTOM** is sent. 7 **WM_ICONIFY** is sent. 8 **WM_UNICONIFY** is sent. 9 **WM_ALLICONIFY** is sent. 10-15 Unused *ap_gout2* is a bitmap of extra messages supported. Current all bits are unused. *ap_gout3* is a bitmap indicating message behaviour (if a bit is set, the behaviour exists) as follows: **Bit** **Message** 0 **WM_ICONIFY** message gives coordinates. 1-15 Unused *ap_gout4* is currently unused.
AES_OBJECT	13	**AES Extended Objects** ap_gout1 is 0 if 3D objects are not supported or 1 if they are. *ap_gout2* is 0 if **objc_sysvar()** is not present, 1 if **MultiTOS** v1.01 **objc_sysvar()** is present, or 2 if extended **objc_sysvar()** is present. *ap_gout3* is 0 if the system font is the only font supported or 1 if **GDOS** fonts are also supported. *ap_gout4* is reserved for OS extensions.
AES_FORM	14	**AES Form Support** *ap_gout1* is 0 if 'flying dialogs' are not supported or 1 if they are. *ap_gout2* is 0 if keyboard tables are not supported or 1 if Mag!X style keyboard tables are supported. *ap_gout3* is 0 if the last cursor position from **objc_edit()** is not returned or 1 if it is. *ap_gout4* is currently reserved.

BINDING

```
intin[0] = ap_gtype;

crys_if(0x82);

*ap_gout1 = intout[1];
*ap_gout2 = intout[2];
*ap_gout3 = intout[3];
*ap_gout4 = intout[4];

return intout[0];
```

RETURN VALUE

appl_getinfo() returns 1 if an error occurred or 0 otherwise.

VERSION NOTES

Using an *ap_gtype* value of 4 and above is only supported as of **AES** version 4.1.

COMMENTS

Many of the *ap_gtype* return values identify features of **TOS** not supported by Atari but for the benefit of third-party vendors. You should contact the appropriate third-party for documentation on these functions.

SEE ALSO

appl_init()

appl_init()

WORD appl_init(VOID)

appl_init() should be the first function called in any application that intends to use **GEM** calls.

OPCODE

10 (0x0A)

AVAILABILITY

All **AES** versions.

PARAMETERS

The function as prototyped accepts no parameters, however, all ‘C’ compilers use this call to set up internal information as well as to update the applications’ global array.

BINDING

```
return crys_if(0x0A);
```

RETURN VALUE

appl_init() returns the applications’ global identifier if successful or -1 if the AES cannot register the application. If successful, the global identifier should be stored in a global variable for later use.

Besides the return value, the **AES** fills in the application’s global array (to reference the global array see your programming languages’ manual).

Name	*global[x]*	Meaning

_AESversion	0	**AES** version number.
_AESnumapps	1	Number of concurrent applications possible (normally 1). **MultiTOS** will return -1.
_AESapid	2	Application identifier (same as **appl_init()** return value).
_AESappglobal	3-4	**LONG** global available for use by the application.
_AESrscfile	5-6	Pointer to the base of the resource loaded via **rsrc_load()**.
—	7-12	Reserved
_AESmaxchar	13	Current maximum character used by the **AES** to do **vst_height()** prior to writing to the screen. This entry is only present as of **AES** version 0x0400.
_AESminchar	14	Current minimum character used by the **AES** to do **vst_height()** prior to writing to the screen. This entry is only present as of **AES** version 0x0400.

VERSION NOTES See above.

SEE ALSO **appl_exit()**

appl_read()

WORD appl_read(***ap_id, length, message*** **)**
WORD ***ap_id, length*****;**
VOIDP ***message*****;**

appl_read() is designed to facilitate inter-process communication between processes running under the **AES**. The call will halt the application until a message of sufficient length is available (see version notes below).

OPCODE 11 (0x0B)

AVAILABILITY All **AES** versions.

PARAMETERS *ap_id* is your application identifier as returned by **appl_init()**. *length* is the length (in bytes) of the message to read. *message* is a pointer to a memory buffer where the incoming message should be copied to.

BINDING

```
intin[0] = ap_id;
intin[1] = length;

addrin[0] = message;

return crys_if(0x0B);
```

RETURN VALUE **appl_read()** returns 0 if an error occurred or non-zero otherwise.

VERSION NOTES If the **AES** version is 4.0 or higher and **appl_getinfo()** indicates that this feature is supported, *ap_id* takes on an additional meaning. If **APR_NOWAIT** (-1) is passed instead of *ap_id*, **appl_read()** will return immediately if no message is currently waiting.

COMMENTS Normally this call is not used. **evnt_multi()** or **evnt_mesag()** is used instead for standard message reception. **appl_read()** is required for reading messages that are long and/or of variable length.

It is recommended that message lengths in multiples of 16 bytes be used.

SEE ALSO **appl_write()**

appl_search()

WORD appl_search(***mode, fname, type, ap_id*** **)**
WORD ***mode*****;**
CHAR ****fname*****;**
WORD ****type,*ap_id*****;**

appl_search() provides a method of identifying all of the currently running processes.

OPCODE 18 (0x12)

AVAILABILITY Available only in **AES** versions 4.0 and above when **appl_getinfo()** indicates its presence.

PARAMETERS *mode* specifies the search mode as follows:

Name	*mode*	Meaning
APP_FIRST	0	Return the filename of the first process
APP_NEXT	1	Return the filename of subsequent processes

fname should point to a memory location at least 9 bytes long to hold the 8 character process filename found and the **NULL** byte. *type* is a pointer to a **WORD** into which will be placed the process type as follows:

Name	*type*	Meaning
APP_SYSTEM	0x01	System process
APP_APPLICATION	0x02	Application
APP_ACCESSORY	0x04	Accessory
APP_SHELL	0x08	

The *type* parameter is actually a bit mask so it is possible that a process containing more than one characteristic will appear. The currently running shell process (usually the desktop) will return a value of **APP_APPLICATION** | **APP_SHELL** (0x0A).

ap_id is a pointer to a word into which will be placed the processes' application identifier.

BINDING

```
intin[0] = mode;

addrin[0] = fname;
addrin[1] = type;
addrin[2] = ap_id;

return crys_if(0x12);
```

RETURN VALUE

appl_search() returns 0 if no more applications exist or 1 when more processes exist that meet the search criteria.

appl_tplay()

WORD appl_tplay(***mem, num, scale*** **)**
VOIDP ***mem*****;**
WORD ***num, scale*****;**

appl_tplay() plays back events originally recorded with **appl_trecord()**.

OPCODE

14 (0x0E)

AVAILABILITY

All **AES** versions.

PARAMETERS

mem is a pointer to an array of **EVNTREC** structures (see **appl_trecord()**). *num* indicates the number of **EVNTREC**'s to play back.

scale indicates on a scale of 1 to 10000 how fast the **AES** will *attempt* to play back your recording. A value of 100 will play it back at recorded speed. A value of 200 will play the events back at twice the recorded speed, and 50 will play back the events at half of the recorded speed. Other values will respond accordingly.

BINDING

```
intin[0] = num;
intin[1] = scale;

addrin[0] = mem;

return crys_if(0x0E);
```

RETURN VALUE **appl_tplay()** always returns 1 meaning no error occurred.

CAVEATS This function does not work correctly on **AES** versions less than 1.40 without a patch program available from Atari Corp.

SEE ALSO **appl_trecord()**

appl_trecord()

WORD appl_trecord(***mem, num*** **)**
VOIDP ***mem*****;**
WORD ***num*****;**

appl_trecord() records **AES** events for later playback.

OPCODE 15 (0x0F)

AVAILABILITY All **AES** versions.

PARAMETERS *mem* points to an array of *num* **EVNTREC** structures into which the **AES** will record events as indicated here:

```
typedef struct pEvntrec
{
        WORD ap_event;
        LONG ap_value;
} EVNTREC;
```

ap_event defines the required interpretation of *ap_value* as follows:

Name	*ap_event*	Event	*ap_value*
APPEVNT_TIMER	0	Timer	Elapsed Time (in milliseconds)
APPEVNT_BUTTON	1	Button	low word = state (1 = down) high word = # of clicks
APPEVNT_MOUSE	2	Mouse	low word = X pos high word = Y pos
APPEVNT_KEYBOARD	3	Keyboard	bits 0-7: ASCII code bits 8-15: scan code bits 16-31: shift key state

BINDING

```
intin[0] = num;

addrin[0] = mem;

return crys_if(0x0F);
```

RETURN VALUE **appl_trecord()** returns the number of events actually recorded.

CAVEATS This function does not work correctly on **AES** versions less than 1.40 without a patch program available from Atari Corp.

SEE ALSO **appl_tplay()**

appl_write()

WORD appl_write(***ap_id, length, msg*** **)**
WORD ***ap_id, length*****;**
VOIDP ***msg*****;**

appl_write() can be used to send a message to a valid message pipe.

OPCODE 12 (0x0C)

AVAILABILITY All **AES** versions.

PARAMETERS *ap_id* is the application identifier of the process to which you wish to send the message. *length* specifies the number of bytes present in the message. *msg* is a pointer to a memory buffer with at least *length* bytes available.

BINDING

```
intin[0] = ap_id;
intin[1] = length;

addrin[0] = msg;

return crys_if(0x0C);
```

RETURN VALUE **appl_write()** returns 0 if an error occurred or greater than 0 if the message was sent successfully.

VERSION NOTES As of **AES** version 1.40, desk accessories may send **MN_SELECTED** messages to the desktop to trigger desktop functions.

As of **AES** version 4.00 you can use **shel_write(**7,...**)** to 'broadcast' a message to all processes running with the exception of the **AES** itself, the desktop, and your own application. See **shel_write()** for details.

COMMENTS It is recommended that you always send messages in 16 byte blocks using a **WORD** array of 8 elements as the **AES** does.

SEE ALSO **appl_read()**, **shel_write()**

Event Library

The *Event Library* consists of a group of system calls which are used to monitor system messages including mouse clicks, keyboard usage, menu bar interaction, timer calls, and mouse tracking. The library consists of the following calls:

- **evnt_button()**
- **evnt_dclick()**
- **evnt_keybd()**
- **evnt_mesag()**
- **evnt_mouse()**
- **evnt_multi()**
- **evnt_timer()**
- **evnt_button()**

evnt_button()

WORD evnt_button(***clicks, mask, state, mx, my, button, kstate*** **)**
WORD ***clicks, mask, state*****;**
WORD ****mx, *my, *button, *kstate*****;**

evnt_button() releases control to the operating system until the specified mouse button event has occurred.

OPCODE 21 (0x15)

AVAILABILITY All **AES** versions.

PARAMETERS *clicks* specifies the number of mouse-clicks that must occur before returning. *mask* specifies the mouse buttons to wait for as follows:

Name	*mask*	Meaning
LEFT_BUTTON	0x01	Left mouse button
RIGHT_BUTTON	0x02	Right mouse button
MIDDLE_BUTTON	0x04	Middle button (this button would be the first button to the left of the rightmost button on the device).
—	0x08 . .	Other buttons (0x08 is the mask for the button to the immediate left of the middle button. Masks continue leftwards).

state specifies the button state that must occur before returning as follows:

mask	Meaning
0x00	All buttons released
0x01	Left button depressed
0x02	Right button depressed
0x04	MIddle button depressed
0x08 . .	etc...

mx is a pointer to a **WORD** which upon return will contain the x-position of the mouse pointer at the time of the event. *my* is a pointer to a **WORD** which upon return will contain the y-position of the mouse pointer at the time of the event.

button is a pointer to a **WORD** which upon return will contain the mouse button state as defined in *state*.

kstate is a pointer to a **WORD** which upon return will contain the current status

of the keyboard shift keys. The value is a bit-mask defined as follows:

Name	Mask	Key
K_RSHIFT	0x01	Right Shift
K_LSHIFT	0x02	Left Shift
K_CTRL	0x04	Control
K_ALT	0x08	Alternate

BINDING

```
intin[0] = clicks;
intin[1] = mask;
intin[2] = state;

crys_if(0x15);

*mx = intout[1];
*my = intout[2];
*button = intout[3];
*kstate = intout[4];

return intout[0];
```

RETURN VALUE

Upon exit, **evnt_button()** returns a **WORD** indicating the number of times the mouse button state matched *state*.

COMMENTS

A previously undocumented feature of this call is accessed by logically OR'ing the *mask* parameter with 0x100. This causes the call to return when independent buttons are depressed. For example, a *mask* value of 0x03 will return when both the left and right mouse buttons are depressed. A *mask* value of 0x103 will cause the call to return when either button is depressed.

SEE ALSO

evnt_multi()

evnt_dclick()

WORD evnt_dclick(*new*, *flag*)
WORD *new*, *flag*;

evnt_dclick() sets the mouse double-click response rate. This call is global, and thus, affects all applications.

OPCODE

26 (0x1A)

AVAILABILITY

All **AES** versions.

PARAMETERS

If *flag* is **EDC_INQUIRE** (0), *new* is ignored and the current double-click rate is returned. If *flag* is **EDC_SET** (1), *new* specifies the new double-click rate as

follows:

flag	Response
0	Slowest
1	
2	
3	
4	Fastest

BINDING

```
intin[0] = new;
intin[1] = flag;

return crys_if(0x1A);
```

RETURN VALUE

evnt_dclick() returns the newly set or current double-click rate based on *flag*.

COMMENTS

Because this setting is global for all applications, Atari has strongly recommended that developers use this call *only* where appropriate (such as in a configuration CPX like the General Setup CPX included with **XCONTROL**).

evnt_keybd()

WORD evnt_keybd(VOID)

evnt_keybd() relinquishes program control to the operating system until a valid keypress is available in the applications' message pipe.

OPCODE

20 (0x14)

AVAILABILITY

All **AES** versions.

PARAMETERS

None

BINDING

```
return crys_if(0x14);
```

RETURN VALUE

evnt_keybd() returns a 16-bit value containing the ASCII code of the key entered in the lower eight bits and the scan code in the upper 8-bits.

VERSION NOTES

TOS versions released at or above 2.06 and 3.06 disabled reception of keys 1 through 9 on the numeric keypad when used in conjunction with the alternate key. Users may now enter the full range of ASCII values by holding down ALT, typing in the decimal ASCII code, and then releasing the ALT key. These keys, therefore, should not be used by applications. The standard numeric keypad is still available.

SEE ALSO

evnt_multi()

evnt_mesag()

WORD evnt_mesag(*msg*)
WORD **msg*;

evnt_mesag() releases control to the operating system until a valid system message is available in the applications' message pipe.

OPCODE 23 (0x17)

AVAILABILITY All **AES** versions.

PARAMETERS *msg* is a pointer to an array of 8 **WORD'**s to be used as a message buffer.

BINDING

```
addrin[0] = msg
return crys_if(0x17);
```

RETURN VALUE The return value is currently reserved by Atari and currently is defined as 1. The array *msg* is filed in with the following values:

Index	Description	Possible Values	#
msg[0]	Message Type	**MN_SELECTED**	10
		WM_REDRAW	20
		WM_TOPPED	21
		WM_CLOSED	22
		WM_FULLED	23
		WM_ARROWED	24
		WM_HSLID	25
		WM_VSLID	26
		WM_SIZED	27
		WM_MOVED	28
		WM_UNTOPPED	30
		WM_ONTOP	31
		WM_BOTTOM	33
		WM_ICONIFY	34
		WM_UNICONIFY	35
		WM_ALLICONIFY	36
		WM_TOOLBAR	37
		AC_OPEN	40
		AC_CLOSE	41
		AP_TERM	50
		AP_TFAIL	51
		AP_RESCHG	57
		SHUT_COMPLETED	60
		RESCH_COMPLETED	61
		AP_DRAGDROP	63
		SH_WDRAW	72
		CH_EXIT	90
msg[1]	The application identifier of the sending application.	Any valid *ap_id*.	
msg[2]	The length of the message *beyond* 16 bytes (use **appl_read()** to read the excess).	Currently all system messages return 0 in this slot. Only user-defined messages utilize a higher value.	

Each system message can be interpreted as follows:

Message	Extended Information
MN_SELECTED	A menu item has been selected by the user. *msg[3]* contains the object number of the menu title and *msg[4]* contains the object number of the menu item. As of **AES** version 4.0 (and when indicated by **appl_getinfo()**), *msg[5]* and *msg[6]* contain the high and low word, respectively, of the object tree of the menu item. *msg[7]* contains the parent object index of the menu item.
WM_REDRAW	This message alerts an application that a portion of the screen needs to be redrawn. *msg[3]* contains the handle of the window to redraw. *msg[4-7]* are the x, y, w, and h respectively of the 'dirtied' area. When the message is received the window contents should be drawn (or a representative icon if the window is iconified).
WM_TOPPED	This message is sent when an application window which is currently not the top window is clicked on by the user. *msg[3]* contains the handle of the window. You should use **wind_set(** handle, **WF_TOP**, *msg[3]*, 0, 0, 0**)** to actually cause the window to be topped.
WM_CLOSED	This message is sent when the user clicks on a windows' close box. *msg[3]* contains the handle of the window to close. You should react to this message with **wind_close()**.
WM_FULLED	This message is sent when the user clicks on a windows' full box. If the window is not at full size, the window should be resized using **wind_set(**handle, **WF_CURRXYWH**,... to occupy the entire screen minus the menu bar (see **wind_get()**). If the window was previously 'fulled' and has not been resized since, the application should return the window to its previous size.

<table>
<tr><td>WM_ARROWED</td><td>This message is sent to inform an application that one of its slider gadgets has been clicked on.

A row or column message is sent when a slider arrow is selected. A ‘page’ message is sent when a darkened area of the scroll bar is clicked. This usually indicates that the application should adjust the window’s contents by a larger amount than with the row or column messages.

msg[3] indicates which action was actually selected as follows:

<table><tr><th>Name</th><th>Value</th><th>Meaning</th></tr><tr><td>WA_UPPAGE</td><td>0</td><td>Page Up</td></tr><tr><td>WA_DNPAGE</td><td>1</td><td>Page Down</td></tr><tr><td>WA_UPLINE</td><td>2</td><td>Row Up</td></tr><tr><td>WA_DNLINE</td><td>3</td><td>Row Down</td></tr><tr><td>WA_LFPAGE</td><td>4</td><td>Page Left</td></tr><tr><td>WA_RTPAGE</td><td>5</td><td>Page Right</td></tr><tr><td>WA_LFLINE</td><td>6</td><td>Column Left</td></tr><tr><td>WA_RTLINE</td><td>7</td><td>Column Right</td></tr></table></td></tr>
<tr><td>WM_HSLID</td><td>This message indicates that the horizontal slider has been moved. msg[3] contains the new slider position ranging from 0 to 1000.

Note: Slider position is relative and not related to slider size.</td></tr>
<tr><td>WM_VSLID</td><td>This message indicates that the vertical slider has been moved. msg[3] contains the new slider position ranging from 0 to 1000.

Note: Slider position is relative and not related to slider size.</td></tr>
<tr><td>WM_SIZED</td><td>This message occurs when the user drags the window sizing gadget. msg[3] contains the window handle. msg[4-7] indicate the x, y, w, and h respectively of the new window location.

Use wind_set(handle, WF_CURRXYWH,... to actually size the window.

WM_SIZED and WM_MOVED usually share common handling code.</td></tr>
<tr><td>WM_MOVED</td><td>This message occurs when the user moves the window by dragging the windows’ title bar. msg[3] contains the handle of the window being moved. msg[4-7] indicate the x, y, w, and h respectively of the new window location.

Use wind_set(handle, WF_CURRXYWH,...) to actually move the window.

WM_MOVED and WM_SIZED usually share common handling code.</td></tr>
<tr><td>WM_UNTOPPED</td><td>This message is sent when the current window is sent behind one or more windows as the result of another window being topped. msg[3] contains the handle of the window being untopped.

The application need take no action. The message is for informational use only.</td></tr>
</table>

WM_ONTOP	This message is sent when an applications' window is brought to the front on a multitasking **AES**. *msg[3]* is the handle of the window being brought to the front. This message requires no action, it is for informational purposes only.
WM_BOTTOM	This message is sent when the user shift-clicks on the window's (specified in *msg*[3]) mover bar to indicate that the window should be sent to the bottom of the window stack by using **wind_set()** with a parameter of **WF_BOTTOM**.
WM_ICONIFY	This message is sent when the user clicks on the **SMALLER** window gadget. *msg*[3] indicates the handle of the window to be iconified. *msg*[4-7] indicate the x, y, w, and h of the iconified window. If the iconified window represents a single window this message should be responded to by using **wind_set()** with a parameter of **WF_ICONIFY**.
WM_UNICONIFY	This message is sent when the user double-clicks on an iconified window. *msg*[3] indicates the handle of the window to be iconified. *msg*[4-7] indicate the x, y, w, and h of the original window. This message should be responded to by using **wind_set()** with a parameter of **WF_UNICONIFY**.
WM_ALLICONIFY	This message is sent when the user CTRL-clicks on the **SMALLER** window gadget. *msg*[3] indicates which window's gadget was clicked. *msg*[4-7] indicates the position at which the new iconified window should be placed. The application should respond to this message by closing all open windows and opening a new iconified window at the position indicated which represents the application.
WM_TOOLBAR	This message is sent when a toolbar object is clicked. *msg*[3] contains the handle of the window containing the toolbar. *msg*[4] contains the object index of the object clicked. *msg*[5] contains the number of clicks. *msg*[6] contains the state of the keyboard shift keys at the time of the click (as in **evnt_keybd()**).
AC_OPEN	This message is sent when the user has selected a desk accessory to open. *msg[4]* contains the application identifier (as returned by **appl_init()**) of the accessory to open.
AC_CLOSE	This message is sent to a desk accessory when the accessory should be closed. *msg[3]* is the application identifier (as returned by **appl_init()**) of the accessory to close. Do not close any windows your accessory had open, the system will do this for you. Also, do not require any feedback from the user when this is received. Treat this message as a 'Cancel' from the user.

AP_TERM	This message is sent when the system requests that the application terminate. This is usually the result of a resolution change but may also occur if another application sends this message to gain total control of the system. The application should shut down immediately after closing windows, freeing resources, etc... *msg[5]* indicates the reason for the shut down as follows: **AP_TERM** (50) = Just shut down. **AP_RESCHG** (57) = Resolution Change. If for some reason, your process can not shut down you must inform the **AES** by sending an **AP_TFAIL** (51) message by using **shel_write()** mode 10 (see **shel_write()**). Note: Desk Accessories wil always be sent **AC_CLOSE** messages, not **AP_TERM**.
AP_TFAIL	This message should be sent to the system (see **shel_write()**) when an application has received an **AP_TERM** (50) message and cannot shut down. *msg[0]* should contain **AP_TFAIL** and *msg[1]* should contain the application error code.
AP_RESCHG	This message is actually a sub-command and is only found as a possible value in the **AP_TERM** (50) message (see above).
SHUT_COMPLETED	This message is sent to the application which requested a shutdown when the shutdown is complete and was successful.
RESCH_COMPLETED	This message is sent to an application when a resolution change it requested is completed. *msg[3]* contains 1 if the resolution change was successful and 0 if an error occurred.
AP_DRAGDROP	This message indicates that another application wishes to initiate a drap and drop session. *msg*[3] indicates the handle of the window which had an object dropped on it or -1 if no specific window was targeted. *msg*[4-5] contains the X and Y position of the mouse when the object was 'dropped'. *msg*[6] indicates the keyboard shift state at the time of the drop (as in **evnt_keybd()**). *msg*[7] is a two-byte ASCII packed pipe identifier which gives the file extension of the pipe to open. For more information about the drag & drop protocal, see *Chapter 2:* ***GEMDOS***.
SH_WDRAW	This message is sent to the Desktop to ask it to update an open drive window. *msg*[3] should contain the drive number to update (0 = A:, 1 = B:) or -1 to update all windows.
CH_EXIT	This message is sent when a child process that the application has started, returns. *msg[3]* contains the child's application identifier and *msg[4]* contains its exit code.

VERSION NOTES **WM_UNTOPPED**, **WM_ONTOP**, **AP_TERM**, **AP_TFAIL**, **AP_RESCHG**, **SHUT_COMPLETED**, **RESCH_COMPLETED**, and **CH_EXIT** are new as of

AES version 4.0.

WM_BOTTOM, **WM_ICONIFY**, **WM_UNICONIFY**, **WM_ALLICONIFY**, and **WM_TOOLBAR** are new as of **AES** version 4.1.

No lower version **AES** will send these messages.

The existence (or acceptance) of these messages should also be checked for by using **appl_getinfo()**.

SEE ALSO **evnt_multi()**

evnt_mouse()

WORD evnt_mouse(***flag, x, y, w, h, mx, my, button, kstate*** **)**
WORD ***flag, x, y, w, h*****;**
WORD ****mx, *mx, *button, *kstate*****;**

evnt_mouse() releases control to the operating system until the mouse enters or leaves a specified area of the screen .

OPCODE 22 (0x16)

AVAILABILITY All **AES** versions.

PARAMETERS *flag* specifies the event to wait for as follows:

Name	Value	Meaning
MO_ENTER	0	Wait for mouse to enter rectangle.
MO_LEAVE	1	Wait for mouse to leave rectangle.

The rectangle to watch is specified in *x*, *y*, *w*, *h*. *mx* and *my* are **WORD** pointers which will be filled in with the final position of the mouse.

button is a **WORD** pointer which will be filled in upon return with the final state of the mouse button as defined in **evnt_button()**.

kstate is a **WORD** pointer which will be filled in upon return with the final state of the keyboard shift keys as defined in **evnt_button()**.

BINDING

```
intin[0] = flag;
intin[1] = x;
intin[2] = y;
intin[3] = w;
intin[4] = h;
```

```
crys_if(0x16);

*mx = intout[1];
*my = intout[2];
*button = intout[3];
*kstate = intout[4];

return intout[0];
```

RETURN VALUE The return value of this function is reserved. Currently it always returns 1.

COMMENTS The **evnt_multi()** function can be used to watch two mouse/rectangle events as opposed to one.

SEE ALSO **evnt_multi()**

evnt_multi()

WORD evnt_multi(***events, bclicks, bmask, bstate, m1flag, m1x, m1y, m1w, m1h, m2flag, m2x, m2y, m2w, m2h, msg, locount, hicount, mx, my, ks, kc, mc*** **)**
WORD ***events, bclicks, bmask, bstate, m1flag, m1x, m1y, m1w, m1h, m2flag, m2x, m2y, m2w, m2h*****;**
WORD ****msg*****;**
WORD ***locount, hicount*****;**
WORD ****mx, *my, *ks, *kc, *mc*****;**

evnt_multi() suspends the application until a valid message that the application is interested in occurs. This call combines the functionality of **evnt_button()**, **evnt_keybd()**, **evnt_mesag()**, **evnt_mouse()**, and **evnt_timer()** into one call.

This call is usually the cornerstone of all **GEM** applications that must process system events.

OPCODE 25 (0x19)

AVAILABILITY All **AES** versions.

PARAMETERS *events* is a bit mask which tells the function which events your application is interested in. You should logically ‘OR’ any of the following values together:

Name	Mask	Function
MU_KEYBD	0x01	Wait for a user keypress.
MU_BUTTON	0x02	Wait for the specified mouse button state.
MU_M1	0x04	Wait for a mouse/rectangle event as specified.
MU_M2	0x08	Wait for a mouse/rectangle event as specified.

MU_MESAG	0x10	Wait for a message.
MU_TIMER	0x20	Wait the specified amount of time.

For usage of *bclicks*, *bmask*, *bstate*, *mx*, *my*, *kc*, and *ks*, you should consult **evnt_button()**.

For usage of *m1flag*, *m1x*, *m1y*, *m1w*, *m1h*, *m2flag*, *m2x*, *m2y*, *m2w*, and *m2h*, consult **evnt_mouse()**.

For usage of *msg*, see **evnt_mesag()**.

For usage of *locount* and *hicount*, see **evnt_timer()**.

BINDING

```
intin[0] = events;
intin[1] = bclicks;
intin[2] = bmask;
intin[3] = bstate;
intin[4] = m1flag;
intin[5] = m1x;
intin[6] = m1y;
intin[7] = m1w;
intin[8] = m1h;
intin[9] = m2flag;
intin[10] = m2x;
intin[11] = m2y;
intin[12] = m2w;
intin[13] = m2h;
intin[14] = locount;
intin[15] = hicount;

addrin[0] = msg;

crys_if(0x19);

*mx = intout[1];
*my = intout[2];
*mb = intout[3];
*ks = intout[4];
*kc = intout[5];
*mc = intout[6];

return intout[0];
```

RETURN VALUE

The function returns a bit mask of which events actually happened as in *events*. This may be one or more events and your application should be prepared to handle each.

VERSION NOTES

The only facet of **evnt_multi()** which has changed from **AES** version 4.0 is that which relates to **evnt_mesag()**. For further information you should consult that section.

CAVEATS

Under **TOS** 1.0, calling this function from a desk accessory with the **MU_TIMER**

mask and *locount* and *hicount* being equal to 0 could hang the system.

SEE ALSO **evnt_button(), evnt_keybd(), evnt_mesag(), evnt_mouse(), evnt_timer()**

evnt_timer()

WORD evnt_timer(***locount, hicount*** **)**
WORD ***locount, hicount*****;**

evnt_timer() releases control to the operating system until a specified amount of time has passed.

OPCODE 24 (0x18)

AVAILABILITY All **AES** versions.

PARAMETERS *locount* is the low word of a 32-bit time value specified in milliseconds. *hicount* is the high portion of that 32-bit value.

BINDING

```
intin[0] = locount;
intin[1] = hicount;

return crys_if(0x18);
```

RETURN VALUE The return value is reserved and is currently always 1.

CAVEATS Under **TOS** 1.0, calling this function from a desk accessory with a both parameters having a value of 0 will hang the system.

COMMENTS This function should not be relyed on as an accurate clock. The time specified is used as a minimum time value only and the function will return at some point after that duration has passed.

SEE ALSO **evnt_multi()**

Form Library

The *Form Library* contains utility functions for the use and control of dialog boxes, alert boxes, and user input. The members of the *Form Library* are:

- **form_alert()**
- **form_button()**
- **form_center()**
- **form_dial()**
- **form_do()**
- **form_error()**
- **form_keybd()**

form_alert()

WORD form_alert(***default, alertstr*** **)**
WORD ***default*****;**
CHAR ****alertstr*****;**

form_alert() displays a standardized alert box and returns the user's selection.

OPCODE 52 (0x34)

AVAILABILITY All **AES** versions.

PARAMETERS *default* contains the number of the exit button which is to be made default (1-3). *alertstr* contains a formatted string as follows: "[#][Alert Text][Buttons]".

specifies the icon to display in the alert as follows:

#	Icon Displayed
0	No Icon
1	
2	
3	
4	
5	

'Alert Text' is a text string of as many as 5 lines composed of up to 30 characters each. Each line is separated by a '|' character.

'*Buttons'* is a text string to define as many as 3 buttons up to 10 characters each. If only one button is used, its text may be as long as 30 characters. Again, each button is separated by a '|' character

BINDING

```
intin[0] = default;

addrin[0] = alertstr;

return crys_if(0x34);
```

RETURN VALUE

form_alert() returns a **WORD** indicating which button was used to exit by the user (A possible value of 1-3).

VERSION NOTES

Icons #4-5 are only available as of **AES** version 4.1.

CAVEATS

Several versions of the **AES** have special quirks related to this function. By following the quidelines below you should avoid any difficulty:

1. All **AES** versions below 1.06 have some difficulty formatting alert strings padded with spaces. If you want your alerts to look right on all **AES** versions, do not pad any button or line with spaces with the exception below.

2. Add one space to the end of the longest text line on an alert. This will prevent the right edge from touching the border in some **AES** versions.

form_button()

WORD form_button(***tree, obj, clicks, newobj*** **)**
OBJECT ****tree*****;**
WORD ***obj, clicks, newobj*****;**

form_button() is a utility function designed to aid in the creation of a custom **form_do()** handler.

OPCODE

56 (0x38)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree is a pointer to a valid object tree in memory you wish to process button events for. *obj* is the object index into *tree* which was clicked on and which needs to be processed.

clicks is the number of times the mouse button was clicked.

newobj returns the next object to gain edit focus or 0 if there are no editable objects. If the top bit of *newobj* is set, this indicates that a **TOUCHEXIT** object was double-clicked.

BINDING

```
intin[0] = obj;
intin[1] = clicks;
```

```
addrin[0] = tree;

crys_if(0x38);

*newobj = intout[1];

return intout[0];
```

RETURN VALUE **form_button()** returns a 0 if it exits finding an **EXIT** or **TOUCHEXIT** object selected or 1 otherwise.

COMMENTS To use this function properly, the application should take the following steps:

1. Monitor mouse clicks with **evnt_multi()** or **evnt_button()**.

2. When a click occurs, use **objc_find()** to determine if the click occurred over the object.

3. If so, call **form_button()** with the appropriate values.

This function was not originally documented by Atari. You may have to add bindings for this function to some earlier 'C' compilers.

SEE ALSO **form_do(), form_keybd()**

form_center()

WORD form_center(***tree, x, y, w, h*** **)**
OBJECT ***tree;***
WORD ***x, *y, *w, *h;***

form_center() is used to modify an object's coordinates so that it will appear in the center of the display screen.

OPCODE 54 (0x36)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* points to a valid **OBJECT** structure (see discussion of resources) which the application wishes to have centered. *x*, *y*, *w*, and *h*, return a clipping rectangle suitable for use in **objc_draw()**.

BINDING

```
addrin[0] = tree;

crys_if(0x36);
```

```
*x = intout[1];
*y = intout[2];
*w = intout[3];
*h = intout[4];

return intout[0];
```

RETURN VALUE The return value is currently reserved. Currently it equals 1.

COMMENTS The values that **form_center()** returns in *x*, *y*, *w*, and *h*, are not necessarily the same as the object's. These values take into account negative borders, outlining, and shadowing. This is meant to provide a suitable clipping rectangle for **objc_draw()**

SEE ALSO **objc_draw()**

form_dial()

WORD form_dial(*mode, x1, y1, w1, h1, x2, y2, w2, h2*)
WORD *mode, x1, y1, w1, h1, x2, y2, w2, h2*;

form_dial() is used to reserve and release screen space for dialog usage. In addition, it also optionally provides grow/shrink box effects.

OPCODE 51 (0x33)

AVAILABILITY All **AES** versions.

PARAMETERS *mode* specifies the action to take and the meaning of remaining parameters as follows:

Name	#	Action
FMD_START	0	This mode reserves the screen space for a dialog. *x2*, *y2*, *w2*, and *h2*, contain the coordinates of the dialog to be used (usually obtained through **form_center()**).
FMD_GROW	1	This mode draws an expanding box from the coordinates specified in *x1*, *y1*, *w1*, and *h1* to the coordinates specified in *x2*, *y2*, *w2*, and *h2*. This call is optional and is not required to display a dialog.
FMD_SHRINK	2	This mode draws a shrinking box from the coordinates specified in *x2*, *y2*, *w2*, and *h2* to the coordinates specified in *x1*, *y1*, *w1*, and *h1*. This call is optional and is not required to display a dialog.
FMD_FINISH	3	This mode releases the screen space for a dialog (previously reserved with mode 0). *x2*, *y2*, *w2*, and *h2* contain the coordinates of the space to release. One of the side-effects of this call is a **WM_REDRAW** message sent to any window which the dialog was covering.

BINDING

```
intin[0] = mode;
intin[1] = x1;
intin[2] = y1;
intin[3] = w1;
intin[4] = h1;
intin[5] = x2;
intin[6] = y2;
intin[7] = w2;
intin[8] = h2;

return crys_if(0x33);
```

RETURN VALUE

The function returns 0 is an error occurred or non-zero otherwise.

VERSION NOTES

The **AES** does not currently make use of mode **FMD_START**. The call should, however, still be executed for upward compatibility.

SEE ALSO

graf_growbox(), **graf_shrinkbox()**

form_do()

WORD form_do(***tree, editobj*** **)**
OBJECT * ***tree*****;**
WORD ***editobj*****;**

form_do() provides an automated dialog handling function to the calling application. It suspends program control, handling all radio buttons, selectable objects, etc... until an object with the **TOUCHEXIT** or **EXIT** flag is selected.

OPCODE

50 (0x32)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree is a pointer to a valid object tree (see the discussion on objects in this chapter) which contains a dialog with at least one **EXIT** or **TOUCHEXIT** button or object.

editobj is the object index into tree which specifies the desired initial location of the edit cursor (the object must be flagged as **EDITABLE**). If the form has no text editable fields, you should use 0.

BINDING

```
intin[0] = editobj;

addrin[0] = tree;

return crys_if(0x32);
```

RETURN VALUE

form_do() returns the object index of the **EXIT** or **TOUCHEXIT** button which

was selected. If the object was double clicked, bit 15 will be set. This means that to obtain the actual object number you should mask off the result with 0x7FFF.

form_error()

WORD form_error(*error*)
WORD *error*;

form_error() displays a pre-defined error alert box to the user.

OPCODE 53 (0x35)

AVAILABILITY All **AES** versions.

PARAMETERS *error* specifies a **MS-DOS** error code as follows:

Name	GEMDOS Error #	*error*	Message
FERR_FILENOTFOUND	-33	2	**File Not Found** The application can not find the folder or file that you tried to access.
FERR_PATHNOTFOUND	-34	3	**Path Not Found** The application cannot find the folder or file that you tried to access.
FERR_NOHANDLES	-35	4	**No More File Handles** The application does not have room to open another document. To make room, close any open document that you do not need.
FERR_ACCESSDENIED	-36	5	**Access Denied** An item with this name already exists in the directory, or this item is set to read-only status.
FERR_LOWMEM	-39	8	**Insufficient Memory** There is not enough memory for the application you just tried to run.
FERR_BADENVIRON	-41	10	**Invalid Environment** There is not enough memory for the application you just tried to run.
FERR_BADFORMAT	-42	11	**Invalid Format** There is not enough memory for the application you just tried to run.

FERR_BADDRIVE	-46	15	**Invalid Drive Specification** The drive you specified does not exist.
FERR_DELETEDIR	-47	16	**Attempt To Delete Working Directory** You cannot delete the folder in which you are working.
FERR_NOFILES	-49	18	**No More Files** The application can not find the folder or file that you tried to access.

The **GEMDOS** error number can be translated into a **MS-DOS** code by subtracting 31 from the absolute value of the error code.

BINDING

```
intin[0] = error;

return crys_if(0x35);
```

RETURN VALUE

The function returns the exit button clicked as in **form_alert()**. It is, however, insignifigant as all of the error alerts have only one button.

CAVEATS

Not every **GEMDOS** error code has a matching alert box.

SEE ALSO

form_alert()

form_keybd()

WORD form_keybd(***tree, obj, nextobj, kc, newobj, keyout*** **)**
OBJECT ****tree*****;**
WORD ***obj, nextobj, kc*****;**
WORD ****newobj, *keyout*****;**

form_keybd() processes keyboard input for dialog box control. It handles special keys such as return, escape, tab, etc... It is only of real use if you are writing a customized **form_do()** routine.

OPCODE

55 (0x37)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree points to a valid **OBJECT** tree containing the dialog you wish to process. *obj* is the object index of the object which currently has edit focus (0 if none). *nextobj* is reserved and should be 1.

kc is the value returned from **evnt_keybd()** or **evnt_multi()** which represents the keypresses' scan code and ASCII value.

newobj is a **WORD** pointer which is filled in on function exit to be the new object with edit focus unless the RETURN key was pressed with a default object present in which case it equals the object index of the object that was the default.

keyout is the value ready to be passed on to **objc_edit()** if no processing was required or 0 if the key was processed and handled by the call.

BINDING

```
intin[0] = obj;
intin[1] = nextobj;
intin[2] = kc;

addrin[0] = tree;

crys_if(0x37);

*newobj = intout[1];
*keyout = intout[2];

return intout[0];
```

RETURN VALUE

form_keybd() returns 0 if a default **EXIT** object was triggered by this call or 1 if the dialog should continue to be processed.

COMMENTS

This function was not originally documented by Atari. You may need to add bindings for this function into some older 'C' compilers.

SEE ALSO

objc_edit(), **form_do(), form_button()**

File Selector Library

The *File Selector Library* contains two functions for displaying the system file selector (or currently installed alternate file selector) and prompting the user to select a file. The members of this library are:

- **fsel_exinput()**
- **fsel_input()**

fsel_exinput()

WORD fsel_exinput(***path, file, button, title*** **)**
CHAR ***path, *file*;**
WORD ***button*;**
CHAR ***title*;**

fsel_exinput() displays the system file selector and offers the user an opportunity to choose a complete **GEMDOS** path specification.

OPCODE

91 (0x5B)

AVAILABILITY

Available from **AES** version 1.40.

PARAMETERS

path should be a pointer to a character buffer at least 128 bytes long (applications wishing to access CD-ROM's should allocate at least 200 bytes). On input the buffer should contain a complete **GEMDOS** path specification including a drive specifier, path string, and wildcard mask as follows: 'drive:\path\mask'. The mask can be any valid **GEMDOS** wildcard (usually *.*).

On function exit, *path* contains final path of the selected file (you will have to strip the mask).

file should point to a character buffer 13 bytes long (12 character filename plus **NULL**). On input its contents will be placed on the filename line of the selector (usually this value can simply be a empty string). On function exit, *file* contains the filename which the user selected.

button is a short pointer which upon function exit will contain **FSEL_CANCEL** (0) if the user selected **CANCEL** or **FSEL_OK** (1) if **OK**.

title should be a pointer to a character string up to 30 characters long which contains the title to appear in the file selector (usually indicates which action the user is about to take).

BINDING

```
addrin[0] = path;
addrin[1] = file;
addrin[2] = label;

crys_if(0x5B);

*button = intout[1];

return intout[0];
```

RETURN VALUE

fsel_exinput() returns 0 if an error occured and 1 otherwise.

VERSION NOTES Some 'C' compilers (Lattice for example) provide a special function which allows **fsel_exinput()** to be used even on earlier **AES** versions.

COMMENTS The path parameter to this function should be validated to ensure that the path actually exists prior to calling this function to prevent confusing the user.

This call should always be used as opposed to **fsel_input()** when it is available. Otherwise, the user has no reminder as to what function s/he is actually undertaking.

SEE ALSO **fsel_input()**

fsel_input()

WORD fsel_input(***path, file, button*** **)**
CHAR ***path, *file*;**
WORD ***button*;**

fsel_input() displays the system file selector and allows the user to select a valid **GEMDOS** path and file.

OPCODE 90 (0x5A)

AVAILABILITY All **AES** versions.

PARAMETERS All parameters are consistent with **fsel_exinput()** with the notable lack of *title*.

BINDING

```
addrin[0] = path;
addrin[1] = file;

crys_if(0x5A);

*button = intout[1];

return intout[0];
```

RETURN VALUE **fsel_input()** returns a 0 if an error occurred or 1 otherwise.

COMMENTS You should never use this function in place of **fsel_exinput()** when **fsel_exinput()** is available.

SEE ALSO **fsel_exinput()**

Graphics Library

The *Graphics Library* provides applications with a variety of utility functions which serve to provide common screen effects, mouse control, and the obtaining of basic screen attributes. The functions of the *Graphics Library* are as follows:

- **graf_dragbox()**
- **graf_growbox()**
- **graf_handle()**
- **graf_mkstate()**
- **graf_mouse()**
- **graf_movebox()**
- **graf_rubberbox()**
- **graf_shrinkbox()**
- **graf_slidebox()**
- **graf_watchbox()**

graf_dragbox()

WORD graf_dragbox(***w, h, sx, sy, bx, by, bw, bh, endx, endy*** **)**
WORD ***w, h, sx, sy, bx, by, bw, bh*****;**
WORD ****endx, *endy*****;**

graf_dragbox() allows the user to move a box frame within the constraints of a bounding rectangle. This call is most often used to give the user a visual 'clue' when an object is being moved on screen.

OPCODE

71 (0x47)

AVAILABILITY

All **AES** versions.

PARAMETERS

w and *h* specify the initial width and height of the box to draw. *sx* and *sy* specify the starting *x* and *y* screen coordinates.

bx, *by*, *bw*, and *bh*, give the coordinates of the bounding rectangle.

endx and *endy* are **WORD** pointers which, on function exit, will be filled in with the ending x and y position of the box.

BINDING

```
intin[0] = w;
intin[1] = h;
intin[2] = sx;
intin[3] = sy;
intin[4] = bx;
intin[5] = by;
intin[6] = bw;
intin[7] = bh;

crys_if(0x47);

*endx = intout[1];
*endy = intout[2];

return intout[0];
```

RETURN VALUE

graf_dragbox() returns a 0 if an error occurred during execution or greater than zero otherwise.

COMMENTS

This call should be made only when the mouse button is depressed. The call returns when the mouse button is released.

SEE ALSO

graf_slidebox()

graf_growbox()

WORD graf_growbox(***x1, y1, w1, h1, x2, y2, w2, h2*** **)**
WORD ***x1, y1, w2, h2, x2, y2, w2, h2*****;**

graf_growbox() is used to provide a visual 'clue' to a user by animating an outline of a box from one set of coordinates to another. It is the complement function to **graf_shrinkbox()**.

OPCODE 73 (0x49)

AVAILABILITY All **AES** versions.

PARAMETERS *x1*, *y1*, *w1*, and *h1* are the screen coordinates of the starting rectangle (where the outline will grow from).

x2, *y2*, *w2*, and *h2* are the screen coordinates of the ending rectangle (where the outline will grow to).

BINDING

```
intin[0] = x1;
intin[1] = y1;
intin[2] = w1;
intin[3] = h1;
intin[4] = x2;
intin[5] = y2;
intin[6] = w2;
intin[7] = h2;

return crys_if(0x49);
```

RETURN VALUE **graf_growbox()** returns 0 if an error occured or non-zero otherwise.

CAVEATS There is currently no defined method of handling an error generated by this function.

COMMENTS This function is what is called by **GEM**'s **form_dial(FMD_GROW**,...

SEE ALSO **form_dial(), graf_shrinkbox()**

graf_handle()

WORD graf_handle(***wcell, hcell, wbox, hbox*** **);**
WORD ****wcell, *hcell, *wbox, *hbox*****;**

graf_handle() returns important information regarding the physical workstation

currently in use by the **AES**.

OPCODE 77 (0x4D)

AVAILABILITY All **AES** versions.

PARAMETERS *wcell* and *hcell* are **WORD** pointers which on function exit will be filled in with the width and height, respectively, of the current system character set.

wbox and *hbox* are **WORD** pointers which on function exit will be filled in with the width and height, respectively, of the minimum bounding box of a **BOXCHAR** character.

BINDING

```
crys_if(0x4D);

*charw = intout[1];
*charh = intout[2];
*boxw = intout[3];
*boxh = intout[4];

return intout[0];
```

RETURN VALUE This function returns the **VDI** handle for the current physical workstation used by the **AES**.

CAVEATS There is currently no defined method of handling an error generated by this function.

COMMENTS The return value of this function is required to open a virtual screen workstation.

SEE ALSO **v_opnvwk()**

graf_mkstate()

WORD graf_mkstate(***mx, my, mb, ks*** **)**
WORD ****mx, *my, *mb, *ks*****;**

graf_mkstate() returns information about the current state of the mouse pointer, buttons, and keyboard shift-key state.

OPCODE 79 (0x4F)

AVAILABILITY All **AES** versions.

PARAMETERS *mx* and my are **WORD** pointers, which, on function exit will be filled in with the current x and y coordinates of the mouse pointer. *mb* is a **WORD** pointer, which,

on function exit will be filled in with the current button state of the mouse as defined in **evnt_button()**.

BINDING

```
crys_if(0x4F);

*mx = intout[1];
*my = intout[2];
*mb = intout[3];
*ks = intout[4];

return intout[0];
```

RETURN VALUE The function return is currently reserved and currently equals 1.

SEE ALSO **evnt_button()**, **vq_mouse()**

graf_mouse()

WORD graf_mouse(***mode*****,** ***formptr*** **)**
WORD ***mode*****;**
VOIDP ***formptr*****;**

graf_mouse() alters the appearance of the mouse form and can be used to hide and display the mouse pointer from the screen.

OPCODE 78 (0x4E)

AVAILABILITY All **AES** versions.

PARAMETERS *mode* is defined as follows:

mode	#	Meaning	Shape
ARROW	0	Change the current mouse cursor shape.	
TEXT_CRSR	1	Change the current mouse cursor shape.	
BUSY_BEE	2	Change the current mouse cursor shape.	
POINT_HAND	3	Change the current mouse cursor shape.	

FLAT_HAND	4	Change the current mouse cursor shape.	
THIN_CROSS	5	Change the current mouse cursor shape.	
THICK_CROSS	6	Change the current mouse cursor shape.	
OUTLN_CROSS	7	Change the current mouse cursor shape.	
USER_DEF	255	Change the current mouse cursor shape.	Form is defined below.
M_OFF	256	Remove the mouse cursor from the screen.	No shape change.
M_ON	257	Display the mouse cursor.	No shape change.
M_SAVE	258	Save the current mouse form in an **AES** provided buffer. Check **appl_getinfo()** for the presence of this feature.	No shape change.
M_LAST	259	Restore the most recently saved mouse form. Check **appl_getinfo()** for the presence of this feature.	Changes the shape as indicated.
M_RESTORE	260	Restore the mouse form to its last shape. Check **appl_getinfo()** for the presence of this feature.	Changes the shape as indicated.

If *mode* is equal to **USER_DEF**, *formptr* must point to a **MFORM** structure as defined below (if *mode* is different than **USER_DEF**, *formptr* should be **NULL**):

```
typedef struct {
        short mf_xhot;
        short mf_yhot;
        short mf_nplanes;
        short mf_fg;
        short mf_bg;
        short mf_mask[16];
        short mf_data[16];
} MFORM;
```

mf_xhot and *mf_yhot* are the location of the mouse ‘hot-spot’. These values should be in the range 0 to 15 and define what offset into the bitmap is actually the ‘point’.

mf_nplanes specifies the number of bit-planes used by the mouse pointer. Currently, the value of 1 is the only legal value.

mf_fg and *mf_bg* are the mask and data colors of the mouse specified as palette

indexes. Usually these values will be 0 and 1 respectively.

mf_mask is an array of 16 **WORD'**s which define the mask portion of the mouse form. *mf_data* is an array of 16 **WORD'**s which define the data portion of the mouse form.

As of **AES** 4.0 and beyond, the **AES** may not allow a mouse form to change to benefit another application. If it is absolutely necessary for the application to display its mouse form, logically OR the mode parameter with **M_FORCE** (0x8000) and make the call.

This will force the **AES** to change to your mouse form. It should, however, be done within the scope of a **wind_update()** sequence.

BINDING

```
intin[0] = mode;

addrin[0] = formptr;

return crys_if(0x4E);
```

RETURN VALUE

graf_mouse() returns a 0 if an error occurred or non-zero otherwise.

CAVEATS

There is currently no defined method of handling an error generated by this function.

SEE ALSO

vsc_form()

graf_movebox()

WORD graf_movebox(***bw, bh, sx, sy, ex, ey*** **)**
WORD ***bw, bh, sx, sy, ex, ey*****;**

graf_movebox() animates a moving box between two points on the screen. It is used to give the user a visual 'clue' to an action undertaken by the application.

OPCODE

72 (0x48)

AVAILABILITY

All **AES** versions.

PARAMETERS

bw and *bh* specify the width and height, respectively, of the box to animate. *sx* and *sy* specify the starting coordinates of the box. *ex* and *ey* specify the ending coordinates of the box.

BINDING

```
intin[0] = bw;
intin[1] = bh;
intin[2] = sx;
```

```
intin[3] = sy;
intin[4] = ex;
intin[5] = ey;

return crys_if(0x48);
```

RETURN VALUE The return value is 0 if an error occured or non-zero otherwise.

CAVEATS There is currently no defined method for handling an error generated by this call.

COMMENTS Some older 'C' bindings referred to this call as **graf_mbox()**. If your compiler still uses this call you should update it.

graf_rubberbox()

WORD graf_rubberbox(***bx, by, minw, minh, endw, endh*** **)**
WORD ***bx, by, minw, minh*****;**
WORD ****endw, *endh*****;**

graf_rubberbox() allows the user to change the size of a box outline with a fixed starting point.

OPCODE 70 (0x46)

AVAILABILITY All **AES** versions.

PARAMETERS *bx* and *by* define the fixed upper-left corner of the box to stretch or shrink.

minw and *minh* specify the minimum width and height that the rectangle can be shrunk to.

endw and *endh* are **WORD** pointers which will be filled in with the ending width and height of the box when the mouse button is released.

BINDING

```
intin[0] = bx;
intin[1] = by;
intin[2] = minw;
intin[3] = minh;

crys_if(0x46);

*endw = intout[1];
*endh = intout[2];

return intout[0];
```

RETURN VALUE **graf_rubberbox()** returns 0 if an error occurred or non-zero otherwise.

CAVEATS There is currently no defined method for handling an error generated by this call.

COMMENTS This function should only be entered when the user has depressed the mouse button as it returns when the mouse button is released.

SEE ALSO **graf_dragbox(), graf_slidebox()**

graf_shrinkbox()

WORD graf_shrinkbox(***x1, y1, w1, h1, x2, y2, w2, h2*** **)**
WORD ***x1, y1, w1, h1, x2, y2, w2, h2*****;**

graf_shrinkbox() displays an animated box shrinking from one rectangle to another. It should be used to provide the user with a visual 'clue' to an action. It is the complement function to **graf_growbox()**.

OPCODE 74 (0x4A)

AVAILABILITY All **AES** versions.

PARAMETERS *x1*, *y1*, *w1*, and *h1* are the coordinates of the rectangle to shrink to.

x2, *y2*, *w2*, and *h2* are the coordinates of the rectangle to shrink from.

BINDING

```
intin[0] = x1;
intin[1] = y1;
intin[2] = w1;
intin[3] = h1;
intin[4] = x2;
intin[5] = y2;
intin[6] = w2;
intin[7] = h2;

return crys_if(0x4A);
```

RETURN VALUE The function returns 0 if an error occurred or non-zero otherwise

CAVEATS There is currently no defined method of handling an error from this call.

COMMENTS This function is essentially the same as **form_dial(FMD_SHRINK**,...

SEE ALSO **form_dial(), graf_growbox()**

graf_slidebox()

WORD graf_slidebox(***tree, parent, obj, orient*** **)**
OBJECT ****tree*;**
WORD ***parent, obj,orient*****;**

graf_slidebox() allows the user to slide a child object within the bounds of its parent. It is often used to implement slider controls.

OPCODE

76 (0x4C)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree is pointer to the object tree containing the child and parent objects.

parent is the object index of an object which bounds the movement of the child. *child* is the object index of the object which can be moved within the bounds of *parent*.

orient specifies the orientation of the allowed movement. 0 is horizontal (left-right), 1 is vertical (up-down).

BINDING

```
intin[0] = parent;
intin[1] = child;
intin[2] = orient;

addrin[0] = tree;

return crys_if(0x4C);
```

RETURN VALUE

The function returns a value specifying the relative offset of the child within the parent as a number between 0 and 1000.

COMMENTS

This call can be used easily with sliders built into dialogs by making the slider bar a **TOUCHEXIT** and calling this function when it is clicked. This call should only be made when the mouse button is depressed as it returns when it is released.

SEE ALSO

graf_movebox()

graf_watchbox()

WORD graf_watchbox(***tree, obj, instate, outstate*** **)**
OBJECT ****tree*;**
WORD ***obj, instate, outstate*****;**

graf_watchbox() modifies the given state of a specified object depending on whether the pointer is within the bounds of the object or outside the bounds of the object as long as the left mouse button is held down.

OPCODE

75 (0x4B)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree is a pointer to the **ROOT** object of the tree which contains the object you wish to watch. *obj* is the object index of the object to watch.

instate is the *ob_state* (see **objc_change()**) to apply while the mouse is inside of the bounds of the object.
outstate is the *ob_state* to apply while the mouse is outside of the bounds of the object.

BINDING

```
intin[0] = obj;
intin[1] = instate;
intin[2] = outstate;

addrin[0] = tree;

return crys_if(0x4B);
```

RETURN VALUE

graf_watchbox() returns a 0 if the mouse button was released outside of the object or a 1 if the button was released inside of the object.

COMMENTS

As this call returns when the mouse button is released, it should only be made when the mouse button is depressed. This call is used internally by **form_button()** and **form_do()** and is usually only necessary if you are replacing one of these handlers.

SEE ALSO

form_button()

Menu Library

The *Menu Library* assists in the handling of system menu bars and popup menus. In addition, individual control of menu items can also be handled through these functions. The members of the *Menu Library* are:

- **menu_attach()**
- **menu_bar()**
- **menu_icheck()**
- **menu_ienable()**
- **menu_istart**
- **menu_popup()**
- **menu_register()**
- **menu_settings()**
- **menu_text()**
- **menu_tnormal()**

menu_attach()

WORD menu_attach(*flag, tree, item, mdata*)
WORD *flag*;
OBJECT **tree*;
WORD *item*;
MENU **mdata*;

menu_attach() allows an application to attach, change, or remove a sub-menu. It also allows the application to inquire information regarding a currently defined sub-menu.

OPCODE

37 (0x25)

AVAILABILITY

This function is only available from **AES** version 3.30 and above. In **AES** versions 4.0 and greater, **appl_getinfo()** should be used to determine its exact functionality.

PARAMETERS

flag indicates the action the application desires as follows:

#	Define	Meaning
0	**ME_INQUIRE**	Return information on a sub-menu attached to the menu item designated by *tree* and *item* in *mdata*.
1	**ME_ATTACH**	Attach or change a sub-menu. *mdata* should be initialized by the application. *tree* and *item* should be the **OBJECT** pointer and index to the menu which is to have the sub-menu attached. If *mdata* is **NULLPTR**, any sub-menu attached will be removed.
2	**ME_REMOVE**	Remove a sub-menu. *tree* and *item* should be the **OBJECT** pointer and index to the menu item which a sub-menu was attached to. *mdata* should be **NULLPTR**.

In all cases except **ME_REMOVE**, *mdata* should point to a **MENU** structure as defined here:

```
typedef struct
{
        OBJECT     *mn_tree;
        WORD       mn_menu;
        WORD       mn_item;
        WORD       mn_scroll;
        WORD       mn_keystate;
} MENU;
```

The **MENU** structure members are defined as follows:

Member	Meaning
mn_tree	Points to the **OBJECT** tree of the sub-menu.
mn_menu	Is an index to the parent object of the menu items.
mn_item	Is the starting menu item.
mn_scroll	If **SCROLL_NO** (0), the menu will not scroll. If **SCROLL_YES** (1), and the number of menu items exceed the menu scroll height, arrows will appear which allow the user to scroll selections.
mn_keystate	This member is unused and should be 0 for this call.

BINDING

```
intin[0] = flag;
intin[1] = item;

addrin[0] = tree;
addrin[1] = mdata;

return crys_if(0x25);
```

RETURN VALUE

menu_attach() returns 0 if an error occurred and the sub-menu could not be attached or 1 if the operation was successful.

CAVEATS

AES versions supporting **menu_attach()** less than 4.1 contain a bug which causes the **AES** to crash when changing or removing a sub-menu attachment.

At present, if you wish to attach a scrolling menu, the menu items must be **G_STRING'**s.

COMMENTS

If a menu bar having attachments is removed with **menu_bar**(**NULL**, **MENU_REMOVE**) those attachments are removed by the system and must be reattached with this call if the menu is redisplayed at a later time.

Several recommendations regarding sub-menus should be adhered to:

1. Menu items which will have sub-menus attached to them should be padded with blanks to the end of the menu.
2. Menu items which will have sub-menus attached to them should not have a keyboard equivalent.
3. Sub-menus will display faster if a byte-boundary is specified.
4. Sub-menus will be shifted vertically to align the start object with the main menu item which it is attached to.
5. Sub-menus will always be adjusted to automatically fit on the screen.
6. There can be a maximum of 64 sub-menu attachments per process (attaching a sub-menu to more than one menu item counts as only one attachment).
7. Do not attach a sub-menu to itself.
8. As a user-interface guideline, there should only be one level of sub-menus, though it is possible to have up to four levels currently.
9. **menu_istart()** works only on sub-menus attached with **menu_attach()**.

SEE ALSO **menu_istart()**, **menu_settings()**, **menu_popup()**

menu_bar()

WORD menu_bar(***tree*****,** ***mode*** **)**
OBJECT ****tree***;**
WORD ***mode*****;**

menu_bar() displays a specialized **OBJECT** tree on the screen as the application menu. It can also be used to determine the owner of the currently displayed menu bar in a multitasking **AES**.

OPCODE 30 (0x1E)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* is a pointer to an **OBJECT** tree which has been formatted for use as a system menu (for more information on the **OBJECT** format of a menu see the discussion on objects in this chapter).

mode is a flag indicating the action to take as follows:

Name	*mode*	Meaning
MENU_REMOVE	0	Erase the menu bar specified in *tree*.
MENU_INSTALL	1	Display the menu bar specified in *tree*.
MENU_INQUIRE	-1	Return the **AES** application identifier of the process which owns the currently displayed system menu. *tree* can be set to **NULL**. The **AES** version must be greater than 4.0 and **appl_getinfo()** must indicate that this is feature is supported.

BINDING

```
intin[0] = mode;

addrin[0] = tree;

return crys_if(0x1E);
```

RETURN VALUE If *mode* is **MENU_REMOVE** (0) or **MENU_INSTALL** (1), the return value indicates an error condition where >0 means no error and 0 means an error occurred. In inquiry mode (*mode* = **MENU_INQUIRE** (-1)), **menu_bar()** returns the application identified of the process which owns the currently displayed menu bar.

COMMENTS The safest way to redraw an application's menu bar is to redraw it only if you are

sure it is currently the active menu bar. In a non-multitasking **AES**, this is a certainty, however, in a multitasking **AES** you should first inquire the menu bar's owner within the scope of a **wind_update(BEG_UPDATE)** call to prevent the system from swapping active menu bars while in the process of redrawing.

SEE ALSO **menu_ienable()**, **menu_icheck()**

menu_icheck()

WORD menu_icheck(***tree, obj, check*** **)**
OBJECT ****tree*;**
WORD ***obj, check*****;**

menu_icheck() adds/removes a checkmark in front of a menu item.

OPCODE 31 (0x1F)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of the current menu. *obj* should be the object index of a menu item. If *check* is **UNCHECK** (0), no checkmark will be displayed next to this item whereas if *check* is **CHECK** (1), a checkmark will be displayed.

BINDING

```
intin[0] = obj;
intin[1] = check;

addrin[0] = obj;

return crys_if(0x1F);
```

RETURN VALUE **menu_icheck()** returns 0 if an error occurred or non-zero otherwise.

SEE ALSO **objc_change()**

menu_ienable()

WORD menu_ienable(***tree, obj, flag*** **)**
OBJECT ****tree*;**
WORD ***obj, flag*****;**

menu_ienable() enables/disables menu items.

OPCODE 32 (0x20)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of the menu to alter. *obj* is the object index of the menu item to modify. *flag* should be set to **DISABLE** (0) to disable the item or **ENABLE** (1) to enable it.

BINDING

```
intin[0] = obj;
intin[1] = flag;

addrin[0] = tree;

return crys_if(0x20);
```

RETURN VALUE **menu_icheck()** returns 0 if an error occurred or non-zero otherwise.

SEE ALSO **objc_change()**

menu_istart()

WORD menu_istart(***flag, tree, imenu, item*** **)**
WORD ***flag*****;**
OBJECT *****tree***;**
WORD ***imenu, item*****;**

menu_istart() shifts a sub-menu that is attached to a menu item to align vertically with the specified object in the sub-menu.

OPCODE 38 (0x26)

AVAILABILITY This function is only available with **AES** versions 3.30 and above.

PARAMETERS *flag* should be set to **MIS_SETALIGN** (1) to modify the alignment of a sub-menu and its parent menu item. If *flag* is set to **MIS_GETALIGN** (0), no modifications will be made, however the sub-menu item index which is currently aligned with its parent menu item is returned.

tree points to the object tree of the menu to alter. *imenu* specifies the object within the submenu which will be aligned with menu item *item*.

BINDING

```
intin[0] = flag;
intin[1] = imenu;
intin[2] = item;

addrin[0] = tree;

return crys_if(0x26);
```

RETURN VALUE **menu_istart()** returns 0 if an error occurred or the positive object index of the sub-menu item which is currently aligned with its parent menu item.

COMMENTS Generally, a sub-menu is aligned so that the currently selected sub-menu item is aligned with its parent menu.

SEE ALSO **menu_attach()**

menu_popup()

WORD menu_popup(***menu, xpos, ypos, mdata*** **)**
MENU ***menu*****;**
WORD ***xpos, ypos*****;**
MENU ***menu*****;**

menu_popup() displays a popup menu and returns the user's selection.

OPCODE 36 (0x24)

AVAILABILITY This function is only available with **AES** versions 3.30 and above.

PARAMETERS *menu* points to a **MENU** structure (defined under **menu_attach()**) containing the popup menu. *xpos* and *ypos* specify the location at which the upper-left corner of the starting object will be placed.

If the function returns a value of 1, the **MENU** structure pointed to by *mdata* will be filled in with the ending state of the menu (including the object the user selected).

As of **AES** version 4.1, if *menu.mn_scroll* is set to **SCROLL_LISTBOX** (-1) when this function is called, a drop-down list box will be displayed instead of a popup menu.

Drop-down list boxes will only display a scroll bar if at least eight entries exist. If you want to force the scroll bar to appear, pad the object with empty **G_STRING** objects with their **DISABLED** flag set.

BINDING

```
intin[0] = xpos;
intin[1] = ypos;

addrin[0] = menu;
addrin[1] = mdata;

return crys_if(0x24);
```

RETURN VALUE **menu_popup()** returns 0 if an error occurred or 1 if successful.

SEE ALSO **menu_attach(), menu_settings()**

menu_register()

WORD menu_register(*ap_id, title*)
WORD *ap_id*;
char **title*;

menu_register() registers desk accessories in the 'Desk' menu and renames **MultiTOS** applications which appear there.

OPCODE 35 (0x23)

AVAILABILITY All **AES** versions.

PARAMETERS *ap_id* specifies the application identifier of the application to register. *title* points to a **NULL**-terminated string containing the title which is to appear in the 'Desk' menu for the accessory or application.

If *ap_id* is set to **REG_NEWNAME** (-1) then the process name given in *title* will be used as the new process name. The new process name should be exactly eight characters terminated with a **NULL**. Pad the string with space characters if necessary.

BINDING

```
intin[0] = ap_id;
addrin[0] = title;
return crys_if(0x23);
```

RETURN VALUE **menu_register()** returns a -1 if an error occurred or the menu identifier otherwise.

VERSION NOTES Applications other than desk accessories should not call this function unless they are running under **MultiTOS**.

COMMENTS Desk accessories should store the return value as this is the value that will be included with future **AC_OPEN** messages to identify the accessory.

Applications running under **MultiTOS** may use this function to provide a more functional title for the 'Desk' menu than the program's filename.

Calling **menu_register()** with a parameter of **REG_NEWNAME** is used to change the internal process name of the application returned by **appl_find()** and **appl_search()**. This is useful if you know another process will attempt to find your

application as a specific process name and the user may have renamed your application filename (normally used as the process name).

menu_settings()

WORD menu_settings(*flag*, *set*)
WORD *flag*;
MN_SET **set*;

menu_settings() changes the global settings for popup and scrollable menus.

OPCODE

39 (0x27)

AVAILABILITY

This function is only available with **AES** versions 3.30 and above.

PARAMETERS

If *flag* is 0, current settings are read into the **MN_SET** structure pointed to by *set*. If *flag* is 1, current settings are set from the **MN_SET** structure pointed to by *set*. **MN_SET** is defined as follows:

```
typedef struct
{
        /* Submenu-display delay in milliseconds */
        LONG display;

        /* Submenu-drag delay in milliseconds */
        LONG drag;

        /* Single-click scroll delay in milliseconds*/
        LONG delay;

        /* Continuous-scroll delay in milliseconds */
        LONG speed;

        /* Menu scroll height (in items) */
        WORD height;
} MN_SET;
```

BINDING

```
intin[0] = flag;

addrin[0] = set;

return crys_if(0x27);
```

RETURN VALUE

menu_settings() always returns 1.

COMMENTS

The defaults set by **menu_settings()** are global and not local to an application. You should therefore limit your use of this function to system applications like CPX's and so forth.

menu_text()

WORD menu_text(***tree, obj, text*** **)**
OBJECT ****tree*;**
WORD ***obj*****;**
char ****text*;**

menu_text() changes the text of a menu item.

OPCODE 34 (0x22)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of the menu bar. *obj* specifies the object index of the menu item to change. *text* points to a **NULL**-terminated character string containing the new text.

BINDING

```
intin[0] = obj;

addrin[0] = tree;
addrin[1] = text;

return crys_if(0x22);
```

RETURN VALUE **menu_text()** returns a 0 if an error occurred or non-zero otherwise.

COMMENTS The new menu item text must be no larger than the original menu item text.

menu_tnormal()

WORD menu_tnormal(***tree, obj, flag*** **)**
OBJECT ****tree*;**
WORD ***obj, flag*****;**

menu_tnormal() highlights/un-highlights a menu-title.

OPCODE 33 (0x21)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of the menu. *obj* specifies the object index of the title to change. *flag* should be set to **HIGHLIGHT** (0) to display the title in reverse (highlighted) or **UNHIGHLIGHT** (1) to display it normally.

BINDING

```
intin[0] = obj
intin[1] = flag

addrin[1] = tree

return crys_if(0x21);
```

RETURN VALUE

menu_tnormal() returns 0 if an error occurred or non-zero otherwise.

COMMENTS

This call is usually called by an application after a **MN_SELECTED** message is received and processed to return the menu title to normal.

Object Library

The *Object Library* is responsible for the drawing and manipulation of **AES** objects such as boxes, strings, icons, etc. See earlier in this chapter for a complete discussion of **AES** objects. The *Object Library* includes the following functions:

- **objc_add()**
- **objc_change()**
- **objc_delete()**
- **objc_draw()**
- **objc_edit()**
- **objc_find()**
- **objc_offset()**
- **objc_order()**
- **objc_sysvar()**

objc_add()

WORD objc_add(***tree, parent, child*** **)**
OBJECT ****tree***;**
WORD ***parent, child*****;**

objc_add() establishes a child object's relationship to its parent.

OPCODE 40 (0x28)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree to modify. *parent* and *child* specify the parent and child object to update.

BINDING

```
intin[0] = parent;
intin[1] = child;

addrin[0] = tree;

return crys_if(0x28);
```

RETURN VALUE **objc_add()** returns a 0 if an error occurred or non-zero otherwise.

COMMENTS In order for this function to work, the object to be added must be already be a member of the **OBJECT** array. This function simply updates the *ob_next*, *ob_head*, and *ob_tail* structure members of **OBJECT**s in the object tree. These fields should be initialized to **NIL** (0) in the child to be added.

SEE ALSO **objc_order(), objc_delete()**

objc_change()

WORD objc_change(***tree, obj, rsvd, ox, oy, ow, oh, newstate, drawflag*** **)**
OBJECT ****tree***;**
WORD ***obj, rsvd, ox, oy, ow, oh, newstate, drawflag*****;**

objc_change() changes the display state of an object.

OPCODE 47 (0x2F)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of the object to modify. *obj* specifies the object to

modify.

rsvd is reserved and should be 0.

ox, *oy*, *ow*, and *oh* specify the clipping rectangle if the object is to be redrawn.

newstate specifies the new state of the object (same as *ob_state*).

If *drawflag* is **NO_DRAW** (0) the object is not redrawn whereas if *drawflag* is **REDRAW** (1) the object is redrawn.

BINDING

```
intin[0] = obj;
intin[1] = rsvd;
intin[2] = ox;
intin[3] = oy;
intin[4] = ow;
intin[5] = oh;
intin[6] = newstate;
intin[7] = drawflag;

addrin[0] = tree;

return crys_if(0x2F);
```

RETURN VALUE

objc_change() returns 0 if an error occurred and non-zero otherwise.

COMMENTS

In general, if not redrawing the object, it is usually quicker to manipulate the object tree directly.

SEE ALSO

objc_draw()

objc_delete()

WORD objc_delete(*tree*, *obj*)
OBJECT **tree*;
WORD *obj*;

objc_delete() removes an object from an object tree.

OPCODE

41 (0x29)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree specifies the object tree of the object to delete. *obj* is the object to be deleted.

BINDING

```
intin[0] = obj;

addrin[0] = tree;
```

```
return crys_if(0x29);
```

RETURN VALUE **objc_delete()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS This function does not move other objects in the tree structure, it simply unlinks the specified object from the object chain by updating the other object's *ob_next*, *ob_head*, and *ob_tail* structure members.

SEE ALSO **objc_add()**

objc_draw()

WORD objc_draw(***tree, obj, depth, ox, oy, ow, oh*** **)**
OBJECT ****tree*;**
WORD ***obj, depth, ox, oy, ow, oh*****;**

objc_draw() renders an **AES** object tree on screen.

OPCODE 42 (0x2A)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree to draw. *obj* specifies the object index at which drawing is to begin.

depth specifies the maximum object depth to draw (a value of 1 searches only first generation objects, a value of 2 searches up to second generation objects, up to a maximum of 7 to search all objects).

ox, *oy*, *ow*, and *oh* specify an **AES** style rectangle which defines the clip rectangle to enforce during drawing.

BINDING

```
intin[0] = obj;
intin[1] = depth;
intin[2] = ox;
intin[3] = oy;
intin[4] = ow;
intin[5] = oh;

addrin[0] = tree;

return crys_if(0x2A);
```

RETURN VALUE **objc_draw()** returns 0 if an error occurred or non-zero otherwise.

objc_edit()

WORD objc_edit(***tree, obj, kc, idx, mode*** **)**
OBJECT ****tree*;**
WORD ***obj, kc*****;**
WORD ****idx*
WORD ***mode*****;**

objc_edit() allows manual control of an editable text field.

OPCODE

46 (0x2E)

AVAILABILITY

All **AES** versions.

PARAMETERS

tree specifies the object tree containing the editable object *obj* to modify. mode specifies the action of the call and the meaning of the other parameters as follows:

mode	Value	Meaning
ED_START	0	Reserved for future use. Do not call.
ED_INIT	1	Display the edit cursor in the object specified. *kc* is ignored. The **WORD** pointed to by *idx* is filled in with the current index of the edit cursor in the field.
ED_CHAR	2	A key has been pressed that needs special processing. *kc* contains the keyboard scan code in the high byte and ASCII code in the low byte. *idx* points to the current index of the text cursor in the field. *idx* will be updated as a result of this call.
ED_END	3	Turn off the text cursor.

BINDING

```
intin[0] = obj;
intin[1] = kc;
intin[2] = *idx;
intin[3] = mode;

addrin[0] = tree;

crys_if(0x2E);

*idx = intout[1];
return intout[0];
```

RETURN VALUE

objc_edit() returns 0 if an error occurred or non-zero otherwise.

COMMENTS

This function is usually used in conjunction with **form_keybd()** in a custom **form_do()** handler.

SEE ALSO **form_keybd()**

objc_find()

WORD objc_find(***tree, obj, depth, ox, oy*** **)**
OBJECT * ***tree*****;**
WORD ***obj, depth, ox, oy*****;**

objc_find() determines which object is found at a given coordinate.

OPCODE 43 (0x2B)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree containing the objects to search. The search starts from object index *obj* forward in the object tree.

depth specifies the depth in the tree to search (a value of 1 searches only first generation objects, a value of 2 searches up to second generation objects, up to a maximum of 7 to search all objects).

ox and *oy* specify the coordinate to search at.

BINDING

```
intin[0] = obj;
intin[1] = depth;
intin[2] = ox;
intin[3] = oy;

addrin[0] = tree;

return crys_if(0x2B);
```

RETURN VALUE **objc_find()** returns the object index of the object found at coordinates (*ox*, *oy*) or -1 if no object is found.

objc_offset()

WORD objc_offset(***tree, obj, ox, oy*** **)**
OBJECT * ***tree*****;**
WORD ***obj*****;**
WORD * ***ox*****, *** ***oy*****;**

objc_offset() calculates the true screen coordinates of an object.

OPCODE 44 (0x2C)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree containing *obj*. The **WORD**s pointed to by *ox* and *oy* will be filled in with the true X and Y screen position of object *obj*.

BINDING

```
intin[0] = obj;

addrin[0] = tree;

crys_if(0x2C);

*ox = intout[1];
*oy = intout[2];

return intout[0];
```

RETURN VALUE **objc_offset()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS The *ob_x* and *ob_y* structure members of objects give an offset from their parent as opposed to true screen location. This call is used to determine a true screen coordinate.

The values returned by **objc_offset()** coupled with the *ob_width* and *ob_height* members do not take into account negative borders, shadowing, or sculpturing. When redrawing an object you are responsible for using these values to and the object's state to compensate for a correct clipping rectangle.

SEE ALSO **objc_draw()**

objc_order()

WORD objc_order(***tree, obj, pos*** **)**
OBJECT ****tree*****;**
WORD ***obj, pos*****;**

objc_order() changes the position of an object relative to other child objects of the same parent.

OPCODE 45 (0x2D)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the object tree of object *obj* which is to be moved. *pos* specifies the new position of the object as follows:

Name	*pos*	Meaning
OO_LAST	-1	Make object the last child.
OO_FIRST	0	Make object the first child.
—	1	Make object the second child.
—	2–	etc...

BINDING

```
intin[0] = obj;
intin[1] = pos;

addrin[0] = tree;

return crys_if(0x2D);
```

RETURN VALUE

objc_order() returns 0 if an error occurred or non-zero otherwise.

COMMENTS

objc_order() does not actually move structure elements in memory. It works by updating the **OBJECT** tree's *ob_head*, *ob_tail*, and *ob_next* fields to 'move' the **OBJECT** in the tree hierarchy.

objc_sysvar()

WORD objc_sysvar(***mode, which, in1, in2, out1, out2*** **)**
WORD ***mode, which, in1, in2*****;**
WORD ****out1, *out2*****;**

objc_sysvar() returns/modifies information about the color and placement of 3D object effects.

OPCODE

48 (0x30)

AVAILABILITY

Available as of **AES** version 3.40.

PARAMETERS

mode determines whether attributes should be read or modified. A value of **SV_INQUIRE** (0) will read the current values whereas a value of **SV_SET** (1) will modify the current values. *which* determines what attribute you wish to read or modify.

When reading values, *in1* and *in2* are unused. The two return values are placed in the **WORD**s pointed to by *out1* and *out2*. When modifying values, *out1* and *out2* are unused. *in1* and *in2* specify the new values for the attribute.

The meanings of the two input/output values referred to as val1 and val2 are as follows:

Name	*which*	Values
LK3DIND	1	If val1 is 1, the text of indicator objects does move when selected, otherwise, if 0, it does not. If val2 is 1, the color of indicator objects does change when selected, otherwise, if 0, it does not.
LK3DACT	2	Same as **LK3DIND** for activator objects.
INDBUTCOL	3	val1 specifies the default color for indicator objects. val2 is unused.
ACTBUTCOL	4	val1 specifies the default color for activator objects. val2 is unused.
BACKGRCOL	5	val1 specifies the default color for background objects. val2 is unused.
AD3DVAL	6	val1 specifies the number of extra pixels on each horizontal side of an indicator or activator object needed to accomodate 3D effects. val2 specifies the number of extra pixels on each vertical side of an indicator or activator object needed to accomodate 3D effects. This setting may only be read, not modified.

BINDING

```
intin[0] = mode;
intin[1] = which;
intin[2] = in1;
intin[3] = in2;

crys_if(0x30);

*out1 = intout[1];
*out2 = intout[2];

return intout[0];
```

RETURN VALUE

objc_sysvar() returns 0 if unsuccessful or non-zero otherwise.

COMMENTS

Applications should not use **objc_sysvar()** to change these settings since all changes are global. Only **CPX**s or Desk Accessories designed to modify these parameters should.

Resource Library

The *Resource Library* is responsibe for the loading/unloading of resource files and the manipulation of resource objects in memory. The members of the *Resource Library* are:

- **rsrc_free()**
- **rsrc_gaddr()**
- **rsrc_load()**
- **rsrc_obfix()**
- **rsrc_rcfix()**
- **rsrc_saddr()**

rsrc_free()

WORD rsrc_free(VOID)

rsrc_free() releases memory allocated by **rsrc_load()** for an application's resource.

OPCODE 111 (0x6F)

AVAILABILITY All **AES** versions.

BINDING

```
return crys_if(0x6F);
```

RETURN VALUE **rsrc_free()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS **rsrc_free()** should be called before an application which loaded a resource using **rsrc_load()** exits.

SEE ALSO **rsrc_load()**

rsrc_gaddr()

WORD rsrc_gaddr(*type, index, addr*)
WORD *type, index*;
VOIDPP *addr*;

rsrc_gaddr() returns the address of an object loaded with **rsrc_load()**.

OPCODE 112 (0x70)

AVAILABILITY All **AES** versions.

PARAMETERS The pointer pointed to by *addr* will be filled in with the address of the *index*th resource object of type *type*. Valid values for *type* are as follows:

Name	*type*	Resource Object
R_TREE	0	Object tree
R_OBJECT	1	Individual object
R_TEDINFO	2	**TEDINFO** structure
R_ICONBLK	3	**ICONBLK** structure
R_BITBLK	4	**BITBLK** structure
R_STRING	5	Free String data

R_IMAGEDATA	6	Free Image data
R_OBSPEC	7	*ob_spec* field within **OBJECT**s
R_TEPTEXT	8	*te_ptext* within **TEDINFO**s
R_TEPTMPLT	9	*te_ptmplt* within **TEDINFO**s
R_TEPVALID	10	*te_pvalid* within **TEDINFO**s
R_IBPMASK	11	*ib_pmask* within **ICONBLK**s
R_IBPDATA	12	*ib_pdata* within **ICONBLK**s
R_IBPTEXT	13	*ib_ptext* within **ICONBLK**s
R_BIPDATA	14	*bi_pdata* within **BITBLK**s
R_FRSTR	15	Free string
R_FRIMG	16	Free image

BINDING

```
intin[0] = type;
intin[1] = index;

crys_if(0x70);

*addr = addrout[0];

return intout[0];
```

RETURN VALUE

rsrc_gaddr() returns a 0 if the address in *addr* is valid or non-zero if the object did not exist.

COMMENTS

This function is most often used to obtain the address of **OBJECT** trees, 'free' strings, and 'free' images after loading a resource file.

SEE ALSO

rsrc_saddr()

rsrc_load()

WORD rsrc_load(*fname*)
char **fname*;

rsrc_load() loads and allocates memory for the named resource file.

OPCODE

110 (0x6E)

AVAILABILITY

All **AES** versions.

PARAMETERS

fname is a character pointer to a **NULL**-terminated **GEMDOS** file specification of the resource to load.

BINDING

```
addrin[0] = fname;
```

```
return crys_if(0x6E);
```

RETURN VALUE **rsrc_load()** returns 0 if successful or non-zero if an error occurred.

COMMENTS In addition to loading the resource, all **OBJECT** coordinates are converted from character based coordinates to screen coordinates.

SEE ALSO **rsrc_free()**

rsrc_obfix()

WORD rsrc_obfix(***tree*, *obj*** **)**
OBJECT ****tree*;**
WORD ***obj*;**

rsrc_obfix() converts an object's coordinates from character-based to pixel-based.

OPCODE 114 (0x72)

AVAILABILITY All **AES** versions.

PARAMETERS *tree* specifies the **OBJECT** tree containing the object *obj* to convert.

BINDING

```
intin[0] = obj;
addrin[0] = tree;
return crys_if(0x72);
```

RETURN VALUE **rsrc_obfix()** returns a 0 if successful or non-zero otherwise.

COMMENTS All objects in '.RSC' files have their coordinates based on character positions rather than screen coordinates to allow an object tree to be shown in any resolution. This function converts those character coordinates to pixel coordinates based on the current screen resolution.

SEE ALSO **rsrc_load(), rsrc_rcfix()**

rsrc_rcfix()

WORD rsrc_rcfix(*rc_header*)
VOID *rc_header;

rsrc_rcfix() fixes up coordinates and memory pointers of raw resource data in memory.

OPCODE 115 (0x73)

AVAILABILITY Available only in **AES** versions 4.0 and greater. The presence of this call should also be checked for using **appl_getinfo()**.

PARAMETERS *rc_header* is a pointer to an Atari Resource Construction Set (or compatible) resource file header in memory.

BINDING

```
addrin[0] = rc_header;

return crys_if(0x73);
```

RETURN VALUE **rsrc_rcfix()** returns a 0 if successful or non-zero otherwise.

COMMENTS If a resource has already been loaded with **rsrc_load()** it must be freed by **rsrc_free()** prior to this call. In addition, resources identified with this call must likewise be freed before program termination or another resource file is needed.

SEE ALSO **rsrc_obfix()**

rsrc_saddr()

WORD rsrc_saddr(*type, index, addr*)
WORD *type, index*;
VOID *addr;

rsrc_saddr() sets the address of a resource element.

OPCODE 113 (0x71)

AVAILABILITY All **AES** versions.

PARAMETERS *type* specifies the type of resource element to set as defined under **rsrc_gaddr()**. *index* specifies the index of the element to modify (0 based). *addr* specifies the actual address that will be placed in the appropriate data structure.

BINDING

```
intin[0] = type;
intin[1] = index;

addrin[0] = addr;

return crys_if(0x71);
```

RETURN VALUE

rsrc_saddr() returns 0 if an error occurred or non-zero otherwise.

COMMENTS

In most cases, direct manipulation of the structures involved is quicker and easier than using this call.

SEE ALSO

rsrc_gaddr(), rsrc_load()

Scrap Library

The *Scrap Library* is used to maintain the location of the clipboard directory used for interprocess data exchange. The members of the *Scrap Library* are:

- **scrp_read()**
- **scrp_write()**

scrp_read()

WORD scrp_read(***cpath*** **)**
char ****cpath***;**

scrp_read() returns the location of the current clipboard directory.

OPCODE

80 (0x50)

AVAILABILITY

All **AES** versions.

PARAMETERS

cpath is a pointer to a character buffer of at least 128 bytes into which the clipboard path will be placed.

BINDING

```
addrin[0] = cpath;

return crys_if(0x50);
```

RETURN VALUE

scrp_read() returns 0 if the clipboard path had not been set or non-zero if *cpath* was properly updated.

CAVEATS

The system scrap directory is a global resource. Some programs incorrectly call **scrp_write()** with a path *and* filename when only a pathname should be used. The following is an example of a correctly formatted *cpath* argument:

```
C:\CLIPBRD\
```

Unfortunately, not all programs adhere exactly to this standard. For this reason, programs reading this information from **scrp_read()** should be especially careful that the information returned is parsed correctly. In addition, don't count on a trailing backslash or the existence of a drive specification.

COMMENTS

If a value of 0 is returned and the application wishes to write a scrap to the clipboard you should follow these steps:

- Create a folder '\CLIPBRD\' on the root directory of the user's boot drive ('C:' or 'A:').
- Write your scrap to the directory as 'SCRAP.???' where '???' signifies the type of information contained in the file.
- Allow other applications to access this information by calling **scrp_write()** with the new clipboard path. For example "C:\CLIPBRD\".

A detailed discussion of the proper clipboard data exchange protocol, including information about a scrap directory semaphore useful with **MultiTOS**, is given earlier in this chapter.

SEE ALSO **scrp_write()**

scrp_write()

WORD scrp_write(***cpath*** **)**
char ****cpath***;**

scrp_write() sets the location of the clipboard directory.

OPCODE 81 (0x51)

AVAILABILITY All **AES** versions.

PARAMETERS *cpath* points to a **NULL**-terminated path string containing a valid drive and path specification with a closing backslash. The following is an example of a correctly formatted *cpath* argument:

```
C:\CLIPBRD\
```

BINDING

```
addrin[0] = cpath;

return crys_if(0x51);
```

RETURN VALUE **scrp_write()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS The scrap directory is a global resource. This call should only be used in two circumstances as follows:

- when used to set the default location of the scrap directory using a CPX or accessory at bootup or by the user's request.
- when **scrp_read()** returns an error value and you need to create the clipboard to write information to it.

The clipboard data exchange protocol is discussed in greater detail earlier in this chapter.

SEE ALSO **scrp_read()**

Shell Library

The *Shell Library* contains several miscellaneous functions most often used by the **GEM Desktop** and other 'Desktop-like' applications. Other applications may, however, need specific functions of the *Shell Library* for various tasks. The members of the *Shell Library* are:

- **shel_envrn()**
- **shel_find()**
- **shel_get()**
- **shel_put()**
- **shel_read()**
- **shel_write()**

shel_envrn()

WORD shel_envrn(***value, name*** **)**
char *****value*;**
char ****name*;**

shel_envrn() searches the current environment string for a specific variable.

OPCODE 125 (0x7D)

AVAILABILITY All **AES** versions.

PARAMETERS *value* points to a character pointer which will be filled in with the address of the first character in the environment string following the string given by *name*. If the string given by *name* is not found, *value* will be filled in with **NULL**. For instance, suppose the current environment looked like this:

```
PATH=C:\;D:\;E:\
```

A call made to **shel_envrn()** with *name* pointing to the string 'PATH=' would set the pointer pointed to by *value* to the string 'C:\;D:\;E:\' above.

BINDING

```
addrin[0] = value;
addrin[1] = name;

return crys_if(0x7D);
```

RETURN VALUE **shel_envrn()** currently always returns 1.

VERSION NOTES **AES** versions prior to 1.4 only accepted semi-colons as separators between multiple 'PATH='arguments. Newer versions accept commas as well.

COMMENTS The character string pointed to by name should include the name of the variable *and* the equals sign.

shel_find()

WORD shel_find(***buf*** **)**
char ****buf*;**

shel_find() searches for a file along the **AES**'s current path, any paths specified by the 'PATH' environmental variable, and the calling application's path.

OPCODE 124 (0x7C)

AVAILABILITY All **AES** versions.

PARAMETERS *buf* should point to a character buffer of at least 128 characters and contain the filename of the file to search for on entry. If the function was able to find the file, the buffer pointed to by *buf* will be filled in with the full pathname of the file upon return.

BINDING

```
addrin[0] = buf;

return crys_if(0x7C);
```

RETURN VALUE **shel_find()** returns 0 if the file was not found or non-zero otherwise.

SEE ALSO **shel_write()**

shel_get()

WORD shel_get(***buf, length*** **)**
char ***buf*****;**
WORD ***length*****;**

shel_get() copies the contents of the **AES**'s shell buffer (normally the 'DESKTOP.INF' or 'NEWDESK.INF' file) into the specified buffer.

OPCODE 122 (0x7A)

AVAILABILITY All **AES** versions.

PARAMETERS *buf* points to a buffer at least *length* bytes long into which the **AES** should copy the shell buffer into.

BINDING

```
intin[0] = length;

addrin[0] = buf;

return crys_if(0x7A);
```

RETURN VALUE **shel_get()** returns 0 if an error occurred or non-zero otherwise.

VERSION NOTES **AES** versions prior to version 1.4 had a shell buffer size of 1024 bytes. Versions 1.4 to 3.0 had a shell buffer size of 4192 bytes.

In **AES** versions 4.0 or greater the shell buffer is no longer of a fixed size. When **appl_getinfo()** indicates that this feature is supported, *length* can be specified as **SHEL_BUFSIZE** (-1) to return the size of the current shell buffer.

SEE ALSO **shel_put()**

shel_put()

WORD shel_put(***buf, length*** **)**
char ***buf;***
WORD ***length;***

shel_put() copies information into the **AES**'s shell buffer.

OPCODE 123 (0x7B)

AVAILABILITY All **AES** versions.

PARAMETERS *buf* points to a user memory buffer from which *length* bytes are to be copied into the shell buffer.

BINDING

```
intin[0] = length;
addrin[0] = buf;
return crys_if(0x7B);
```

RETURN VALUE **shel_put()** returns 0 if an error occurred or non-zero otherwise.

VERSION NOTES Prior to **AES** version 4.0 this function would only copy as many bytes as would fit into the current buffer. As of version 4.0, the **AES** will dynamically allocate more memory as needed (up to 32767 bytes) for the shell buffer.

COMMENTS The **Desktop** uses the information in the shell buffer for several purposes. Applications should not use the shell buffer for their own purposes.

SEE ALSO **shel_get()**

shel_read()

WORD shel_read(***name, tail*** **)**
char ***name, *tail;***

shel_read() is used to determine the current application's parent and the command tail used to call it.

OPCODE 120 (0x78)

AVAILABILITY All **AES** versions.

PARAMETERS *name* points to a buffer which upon exit will be filled in with the complete file specification of the application which launched the current process.

tail will likewise be filled in with the initial command line. The first BYTE of the command line indicates the length of the string which actually begins at &*tail[1]*.

BINDING

```
addrin[0] = name;
addrin[1] = tail;

return crys_if(0x78);
```

RETURN VALUE **shel_read()** returns 0 if an error occurred or non-zero otherwise.

CAVEATS **shel_read()** actually returns the arguments to the last **shel_write()** so if a process was **Pexec()**'ed, the information returned will be incorrect.

shel_write()

WORD shel_write(***mode, wisgr, wiscr, cmd, tail*** **)**
WORD ***mode, wisgr, wiscr*****;**
char ****cmd, *tail*****;**

shel_write() is a multi-purpose function which handles the manipulation and launching of processes.

OPCODE 121 (0x79)

AVAILABILITY All **AES** versions. In **AES** versions 4.0 and above, **appl_getinfo()** can be used to determine the highest legal value for *mode* as well as the functionality of extended *mode* bits.

PARAMETERS *mode* specifies the meaning of the rest of the parameters as follows:

Name	*mode*	Meaning
SWM_LAUNCH	0	Launch a **GEM** or **TOS** application or **GEM** desk accessory depending on the extension of the file. This mode is only available as of **AES** version 4.0. *wisgr* is not used in *mode* **SWM_LAUNCH** (0). When the lower eight bits of *mode* are **SWM_LAUNCH** (0), **SWM_LAUNCHNOW** (1), or **SWM_LAUNCHACC** (3), appropriate bits in the upper byte may be set to enter 'extended' mode. The bits in the upper byte are assigned as follows:

Name	Mask	Meaning
SW_PSETLIMIT	0x100	Initial **Psetlimit()**
SW_PRENICE	0x200	Initial **Prenice()**
SW_DEFDIR	0x400	Default Directory
SW_ENVIRON	0x800	Environment

If the upper byte is empty, extended mode is not entered and *cmd* specifies the filename (to search for the file with **shel_find()**) or the complete file specification. Otherwise, if any extended bits are set, *cmd* points to a structure as shown below.

```
typedef struct _shelw
{
      char *newcmd;
      LONG psetlimit;
      LONG prenice;
      char *defdir;
      char *env;
} SHELW;
```

_shelw.newcmd points to the filename formatted in the manner indicated above.

If bit 8 (**SW_PSETLIMIT**) of *mode* is set, *_shelw.psetlimit* contains the maximum memory size available to the process.

If bit 9 of *mode* is (**SW_PRENICE**) set, *_shelw.prenice* contains the process priority of the process to launch.

If bit 10 of *mode* (**SW_DEFDIR**) is set, *_shelw.defdir* points to a character string containing the default directory for the application begin launched.

If bit 11 of *mode* (**SW_ENVIRON**) is set, *_shelw.env* points to a valid environment string for the process.

tail points to a buffer containing the command tail to pass to the process. If *wiscr* is set to **CL_NORMAL** (0), *tail* is passed normally, otherwise, if *wiscr* is set to **CL_PARSE** (1), the **AES** will parse *tail* and set up an **ARGV** environment string.

modes **SWM_LAUNCH** (0), **SWM_LAUNCHNOW** (1), and **SWM_LAUNCHACC** (3) return the **AES** id of the started process. If a 0 is returned, then the process was not launched.

Under **MultiTOS**, processes are launched concurrently with their parent. An exit code is returned in a **CH_EXIT** message when the child terminates. See **evnt_mesag()**.

In **AES** versions 4.0 and above, **appl_getinfo()** should be used to determine the exact result of this call.

SWM_LAUNCHNOW	1	Launch a **GEM** or **TOS** application based on the value of *wisgr*. If *wisgr* is **TOSAPP** (0), the application will be launched as a **TOS** application, otherwise if *wisgr* is **GEMAPP** (1), the application will be launched as a **GEM** application. For the meaning of other parameters, see mode **SWM_LAUNCH** (0). The extended bits in mode are only supported by **AES** versions of at least 4.0. Parent applications which launch children using this mode are suspended under **MultiTOS**. In **AES** versions 4.0 and above, **appl_getinfo()** should be used to determine the exact result of this call.
SWM_LAUNCHACC	3	Launch a **GEM** desk accessory. For the meaning of other parameters, see mode **SWM_LAUNCH** (0). This mode is only supported by **AES** versions of at least 4.0.
SWM_SHUTDOWN	4	Manipulate 'Shutdown' mode. Shutdown mode is usually used prior to a resolution change to cause system processes to terminate. *wisgr*, *cmd*, and *tail* are ignored by this call. The value of *wiscr* determines the action this call takes as follows: **Name** — ***wiscr*** — **Meaning** **SD_ABORT** — 0 — Abort shutdown mode **SD_PARTIAL** — 1 — Partial shutdown mode **SD_COMPLETE** — 2 — Complete shutdown mode During a shutdown, processes which have registered themselves as accepting **AP_TERM** messages will be sent them and all accessories will be sent **AC_CLOSE** messages. In addition, in complete shutdown mode, **AP_TERM** messages will also be sent to accessories. Shutdown mode may be aborted but only by the original caller. The status of the shutdown is sent to the calling processes by **AES** messages. See **evnt_mesag()**. This mode is only supported by **AES** versions greater than or equal to 4.0.
SWM_REZCHANGE	5	Change screen resolution. *wisgr* is the work station ID (same as in **AES** *global[13]*) of the new resolution. No other parameters are utilized. This mode is only recognized as of **AES** version 4.0.
SWM_BROADCAST	7	Broadcast an **AES** message to all processes. *cmd* should point to an 8 **WORD** message buffer containing the message to send. All other parameters are ignored. This mode is only recognized as of **AES** version 4.0.

SWM_ENVIRON	8	Manipulate the **AES** environment. If *wisgr* is **ENVIRON_SIZE** (0), the current size of the environment string is returned. If *wisgr* is **ENVIRON_CHANGE** (1), *cmd* should point to a environment variable to modify. If *cmd* points to "TOSEXT=TOS,TTP", that string will be added. Likewise, "TOSEXT=" will remove that environment variable. If *wisgr* is **ENVIRON_COPY** (2), the **AES** will copy as many as *wiscr* bytes of the current environment string into a buffer pointer to by *cmd.* The function will return the number of bytes *not* copied. This mode is only recognized as of **AES** version 4.0.
SWM_NEWMSG	9	Inform the **AES** of a new message the current application understands. *wisgr* is a bit mask which specifies which new messages the application understands. Currently only bit 0 (**B_UNTOPPABLE**) has a meaning. Setting this bit when calling this function will inform the **AES** that the application understands **AP_TERM** messages. No other parameters are used. This mode is only recognized as of **AES** version 4.0.
SWM_AESMSG	10	Send a message to the **AES**. *cmd* points to an 8 **WORD** message buffer containing the message to send. No other parameters are needed. This mode is only recognized as of **AES** version 4.0.

BINDING

```
intin[0] = mode;
intin[1] = wisgr;
intin[2] = wiscr;

addrin[0] = cmd;
addrin[1] = tail;

return crys_if(0x79);
```

RETURN VALUE

The value **shel_write()** differs depending on the mode which was invoked. See above for details.

VERSION NOTES

Many new features were added as of **AES** version 4.0. For details of each, see above.

Window Library

The *Window Library* is responsible for the displaying and maintenance of **AES** windows. The members of the *Window Library* are:

- **wind_calc()**
- **wind_close()**
- **wind_create()**
- **wind_delete()**
- **wind_find()**
- **wind_get()**
- **wind_new()**
- **wind_open()**
- **wind_set()**
- **wind_update()**

wind_calc()

WORD wind_calc(***request, kind, x1, y1, w1, h1, x2, y2, w2, h2*** **)**
WORD ***request, kind, x1, y1, w1, h1*****;**
WORD ****x2, *y2, *w2, *h2*****;**

wind_calc() returns size information for a specific window.

OPCODE

108 (0x6C)

AVAILABILITY

All **AES** versions.

PARAMETERS

request specifies the mode of this call.

If *request* is **WC_BORDER** (0), *x1*, *y1*, *w1*, and *h1* specify the work area of a window of type *kind*. The call then fills in the **WORD**s pointed to by *x2*, *y2*, *w2*, and *h2* with the full extent of the window.

If request is **WC_WORK** (1), *x1*, *y1*, *w1*, and *h1* specify the full extent of a window of type *kind*. The call fills in the **WORD**s pointed to by *x2*, *y2*, *w2*, and *h2* with the work area of the window.

kind is a bit mask of window 'widgets' present with the window. For a detailed listing of these elements see **wind_create()**.

BINDING

```
intin[0] = request;
intin[1] = kind;
intin[2] = x1;
intin[3] = y1;
intin[4] = w1;
intin[5] = h1;

crys_if(0x6C);

*x2 = intout[1];
*y2 = intout[2];
*w2 = intout[3];
*h2 = intout[4];

return intout[0];
```

RETURN VALUE

wind_calc() returns 0 if an error occurred or non-zero otherwise.

COMMENTS

wind_calc() is unable to calculate correct values when a toolbar is attached to a window. This can be corrected, though, by adjusting the values output by this function with the height of the toolbar.

SEE ALSO

wind_create()

wind_close()

WORD wind_close(*handle*)
WORD *handle*;

wind_close() removes a window from the display screen.

OPCODE 102 (0x66)

AVAILABILITY All **AES** versions.

PARAMETERS *handle* specifies the window handle of the window to close.

BINDING

```
intin[0] = handle;
return crys_if(0x66);
```

RETURN VALUE **wind_close()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS Upon calling **wind_close()** a redraw message for the portion of the screen changed will be sent to all applications.

Calling **wind_close()** does not release the memory allocated to the window structure. **wind_delete()** must be called to permanently destroy the window and free any memory allocated by the **AES** for the window. Until **wind_delete()** is called, the window may be re-opened at any time with **wind_open()**.

SEE ALSO **wind_create(), wind_open(), wind_delete()**

wind_create()

WORD wind_create(*kind, x, y, w, h*)
WORD *kind, x, y, w, h*;

wind_create() initializes a new window structure and allocates any necessary memory.

OPCODE 100 (0x64)

AVAILABILITY All **AES** versions.

PARAMETERS *kind* is a bit array whose elements determine the presence of any ‘widgets’ on the

window as follows:

Name	Mask	Meaning
NAME	0x01	Window has a title bar.
CLOSER	0x02	Window has a close box.
FULLER	0x04	Window has a fuller box.
MOVER	0x08	Window may be moved by the user.
INFO	0x10	Window has an information line.
SIZER	0x20	Window has a sizer box.
UPARROW	0x40	Window has an up arrow.
DNARROW	0x80	Window has a down arrow.
VSLIDE	0x100	Window has a vertical slider.
LFARROW	0x200	Window has a left arrow.
RTARROW	0x400	Window has a right arrow.
HSLIDE	0x800	Window has a horizontal slider.
SMALLER	0x4000	Window has an iconifier.

The parameter kind is created by OR'ing together any desired elements.

x, *y*, *w*, and *h*, specify the maximum extents of the window. Normally this is the entire screen area minus the menu bar (to find this area use **wind_get()** with a parameter of **WF_WORKXYWH**). The area may be smaller to bound the window to a particular size and location.

BINDING

```
intin[0] = kind;
intin[1] = x;
intin[2] = y;
intin[3] = w;
intin[4] = h;

return crys_if(0x64);
```

RETURN VALUE

wind_create() returns a window handle if successful or a negative number if it was unable to create the window.

VERSION NOTES

The **SMALLER** gadget is only available as of **AES** version 4.1.

COMMENTS

A window is not actually displayed on screen with this call, you need to call **wind_open()** to do that.

TOS version 1.00 and 1.02 limited applications to four windows. In **TOS** version 1.04 that limit was raised to seven. As of **MultiTOS** the number of open windows is limited only by memory and the capabilities of an application.

You should ensure that your application calls a **wind_delete()** for each **wind_create()**, otherwise memory may not be deallocated when your application

exits.

SEE ALSO **wind_open(), wind_close(), wind_delete()**

wind_delete()

WORD wind_delete(*handle*)
WORD *handle*;

wind_delete() destroys the specified window and releases any memory allocated for it.

OPCODE 103 (0x67)

AVAILABILITY All **AES** versions.

PARAMETERS *handle* specifies the window handle of the window to destroy.

BINDING

```
intin[0] = handle;

return crys_if(0x67);
```

RETURN VALUE **wind_delete()** returns 0 if an error occurred or non-zero otherwise.

COMMENTS A window should by closed with **wind_close()** before deleting it.

SEE ALSO **wind_create(), wind_open(), wind_close(), wind_new()**

wind_find()

WORD wind_find(*x*, *y*)
WORD *x*, *y*;

wind_find() returns the handle of the window found at the given coordinates.

OPCODE 106 (0x6A)

AVAILABILITY All **AES** versions.

PARAMETERS *x* and *y* specify the coordinates to search for a window at.

BINDING

```
intin[0] = x;
intin[1] = y;
```

```
return crys_if(0x6A);
```

RETURN VALUE **wind_find()** returns the handle of the uppermost window found at location *x*, *y*. If no window is found, the function returns 0 meaning the **Desktop** window.

COMMENTS This function is useful for tracking the mouse pointer and changing its shape depending upon what window it falls over.

wind_get()

WORD wind_get(*handle, mode, parm1, parm2, parm3, parm4*)
WORD *handle, mode*;
WORD **parm1, *parm2, *parm3, *parm4*;

wind_get() returns various information about a window.

OPCODE 104 (0x68)

AVAILABILITY All **AES** versions.

PARAMETERS *handle* specifies the handle of the window to return information about (0 is the desktop window). *mode* specifies the information to return and the values placed into the **WORD**s pointed to by *parm1*, *parm2*, *parm3*, and *parm4* as follows:

Name	*mode*	Meaning
WF_WORKXYWH	4	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with the x, y, w, and h of the current coordinates of the window's work area.
WF_CURRXYWH	5	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with the x, y, w, and h of the current coordinates of the full extent of the window.
WF_PREVXYWH	6	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with the x, y, w, and h of the previous coordinates of the full extent of the window prior to the last **wind_set()** call.
WF_FULLXYWH	7	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with the x, y, w, and h values specified in the **wind_create()** call.
WF_HSLIDE	8	*parm1* is filled in with the current position of the horizontal slider between 1 and 1000. A value of one indicates that the slider is in its leftmost position.
WF_VSLIDE	9	*parm1* is filled in with the current position of the vertical slider between 1 and 1000. A value of one indicates that the slider is in its uppermost position.
WF_TOP	10	*parm1* is filled in with the window handle of the window currently on top. As of **AES** version 4.0 (and when appl_getinfo() indicates), *parm2* is filled in with the owners **AES** id, and *parm3* is filled in with the handle of the window directly below it.

WF_FIRSTXYWH	11	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with the x, y, w, and h of the first **AES** rectangle in the window's rectangle list. If *parm3* and *parm4* are both 0, the window is completely covered.
WF_NEXTXYWH	12	*parm1*, *parm2*, *parm3*, and *parm4* are filled in with subsequent **AES** rectangles for each time this function is called until *parm3* and *parm4* are 0 to signify the end of the list.
WF_NEWDESK	14	As of **AES** versions 4.0 (and when **appl_getinfo()** indicates), this *mode* returns a pointer to the current desktop background **OBJECT** tree. *parm1* contains the high **WORD** of the address and *parm2* contains the low **WORD**.
WF_HSLSIZE	15	*parm1* contains the size of the current slider relative to the size of the scroll bar as a value from 1 to 1000. A value of 1000 indicates that the slider is at its maximum size.
WF_VSLSIZE	16	*parm1* contains the size of the current slider relative to the size of the scroll bar as a value from 1 to 1000. A value of 1000 indicates that the slider is at its maximum size.
WF_SCREEN	17	This *mode* returns a pointer to the current **AES** menu/alert buffer and its size. The pointer's high **WORD** is returned in *parm1* and the pointer's low **WORD** is returned in *parm2*. The length of the buffer is returned as a **LONG** with the upper **WORD** being in *parm3* and the lower **WORD** being in *parm4*. Note that **TOS** 1.02 returns 0 in *w*and *h* by mistake. The menu/alert buffer is used by the **AES** to save the screen area hidden by menus and alert boxes. It is not recommended that applications use this area as its usage is not guaranteed in future versions of the OS.

<table>
<tr><td>**WF_COLOR**</td><td>18</td><td>This *mode* gets the current color of the window widget specified on entry to the function in the **WORD** pointed to by *parm1*. Valid window widget indexes are as follows (**W_SMALLER** is only valid as of **AES** 4.1):

<table>
<tr><th><u>*parm1*</u></th><th><u>Value</u></th><th><u>*ob_type*</u></th></tr>
<tr><td>**W_BOX**</td><td>0</td><td>**IBOX**</td></tr>
<tr><td>**W_TITLE**</td><td>1</td><td>**BOX**</td></tr>
<tr><td>**W_CLOSER**</td><td>2</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_NAME**</td><td>3</td><td>**BOXTEXT**</td></tr>
<tr><td>**W_FULLER**</td><td>4</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_INFO**</td><td>5</td><td>**BOXTEXT**</td></tr>
<tr><td>**W_DATA**</td><td>6</td><td>**IBOX**</td></tr>
<tr><td>**W_WORK**</td><td>7</td><td>**IBOX**</td></tr>
<tr><td>**W_SIZER**</td><td>8</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_VBAR**</td><td>9</td><td>**BOX**</td></tr>
<tr><td>**W_UPARROW**</td><td>10</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_DNARROW**</td><td>11</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_VSLIDE**</td><td>12</td><td>**BOX**</td></tr>
<tr><td>**W_VELEV**</td><td>13</td><td>**BOX**</td></tr>
<tr><td>**W_HBAR**</td><td>14</td><td>**BOX**</td></tr>
<tr><td>**W_LFARROW**</td><td>15</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_RTARROW**</td><td>16</td><td>**BOXCHAR**</td></tr>
<tr><td>**W_HSLIDE**</td><td>17</td><td>**BOX**</td></tr>
<tr><td>**W_HELEV**</td><td>18</td><td>**BOX**</td></tr>
<tr><td>**W_SMALLER**</td><td>19</td><td>**BOXCHAR**</td></tr>
</table>

The *ob_spec* field (containing the color information) used for the object when not selected is returned in the **WORD** pointed to by *parm2*. The *ob_spec* field used for the object when selected is returned in *parm3*.

This *mode* under **wind_get()** is only valid as of **AES** version 3.30. From **AES** versions 4.0 and above, **appl_getinfo()** should be used to determine if this mode is supported.</td></tr>
<tr><td>**WF_DCOLOR**</td><td>19</td><td>This *mode* gets the default color of newly created windows as with **WF_COLOR** above. As above, this mode under **wind_get()** only works as of **AES** version 3.30.

As of **AES** version 4.1, **WF_DCOLOR** changes the color of open windows unless they have had their colors explicitly set with **WF_COLOR**.</td></tr>
<tr><td>**WF_OWNER**</td><td>20</td><td>*parm1* is filled in with the **AES** id of the owner of the specified window. *parm2* is filled in with its open status (0 = closed, 1 = open). *parm3* is filled in with the handle of the window directly above it (in the window order list) and *parm4* is filled in with the handle of the window below it (likewise, in the window order list).

This mode is only available as of **AES** version 4.0 (and when indicated by **appl_getinfo()**).</td></tr>
</table>

WF_BEVENT	24	*parm1*, *parm2*, *parm3*, *parm4* are each interpreted as bit arrays whose bits indicate supported window features. Currently only one bit is supported. If bit 0 of the value returned in *parm1* is 1, that window has been set to be 'un-toppable' and it will never receive **WM_TOPPED** messages, only button clicks. This mode is only available as of **AES** version 4.0 (and when indicated by appl_getinfo()).
WF_BOTTOM	25	*parm1* will be filled in with the handle of the window currently on the bottom of the window list (it may actually be on top if there is only one window). Note also that this does not include the desktop window. This *mode* is only available as of **AES** version 4.0 (and when indicated by **appl_getinfo()**).
WF_ICONIFY	26	*parm1* will be filled in with 0 if the window is not iconified or non-zero if it is. *parm2* and *parm3* contain the width and height of the icon. *parm4* is unused. This *mode* is only available as of **AES** version 4.1 (and when indicated by appl_getinfo()).
WF_UNICONIFY	27	*parm1, parm2, parm3,* and *parm4,* are filled in with the x, y, w, and h of the original coordinates of the iconified window. This *mode* is only available as of **AES** version 4.1 (and when indicated by **appl_getinfo()**).
WF_TOOLBAR	30	*parm1* and *parm2* contain the high and low **WORD** respectively of the pointer to the current toolbar object tree (or **NULL** if none). This *mode* is only available as of **AES** version 4.1.
WF_FTOOLBAR	31	*parm1, parm2, parm3,* are *parm4,* are filled in with the x, y, w, and h, respectively of the first uncovered rectangle of the toolbar region of the window. If *parm3* and *parm4* are 0, the toolbar is completely covered. This *mode* is only available as of **AES** version 4.1.
WF_NTOOLBAR	32	*parm1, parm2, parm3,* and *parm4,* are filled in with the x, y, w, and h, respectively of subsequent uncovered rectangles of the toolbar region. This mode should be repeated to reveal subsequent rectangles until *parm3* and *parm4* are found to be 0. This *mode* is only available as of **AES** version 4.1.

BINDING

```
/* This binding must be different to */
/* accomodate reading WF_COLOR and   */
/* WF_DCOLOR                         */

contrl[0] = 0x68;
contrl[1] = 2;
contrl[2] = 1;
contrl[3] = 0;
contrl[4] = 0;
```

```
intin[0] = handle;
intin[1] = mode;

if(mode == WF_DCOLOR || mode == WF_COLOR)
{
        intin[2] = *x;
        contrl[1] = 3;
}

aes();

*x = intout[1];
*y = intout[2];
*w = intout[3];
*h = intout[4];

return intout[0];
```

RETURN VALUE **wind_get()** returns a 0 if an error occurred or non-zero otherwise.

SEE ALSO **wind_set()**

wind_new()

WORD wind_new(VOID)

wind_new() closes and deletes all of the application's windows. In addition, the state of **wind_update()**, and the mouse pointer hide count is reset.

OPCODE 109 (0x6D)

AVAILABILITY Available as of **AES** version 0x0140.

BINDING

```
return crys_if(0x6D);
```

RETURN VALUE The return value is reserved and currently unused

COMMENTS This function should not be relied upon to clean up after an application. It was designed for parent processes that wish to ensure that a poorly written child process has properly cleaned up after itself.

SEE ALSO **wind_delete(), graf_mouse(), wind_update()**

wind_open()

WORD wind_open(***handle, x, y, w, h*** **)**
WORD ***handle*****;**
WORD x, y, w, h;

wind_open() opens the window specified.

OPCODE 101 (0x65)

AVAILABILITY All **AES** versions.

PARAMETERS *handle* specifies the handle of the window to open as returned by **wind_create()**. *x*, *y*, *w*, and *h* specify the rectangle into which the rectangle should be displayed.

BINDING

```
intin[0] = handle;

return crys_if(0x65);
```

RETURN VALUE **wind_open()** returns a 0 if an error occurred or non-zero otherwise.

COMMENTS This call will also trigger a **WM_REDRAW** message which encompasses the work area of the window so applications should not initially render the work area, rather, wait for the message.

SEE ALSO **wind_close(), wind_create(), wind_delete()**

wind_set()

WORD wind_set(***handle, mode, parm1, parm2, parm3, parm4*** **)**
WORD ***handle, mode, parm1, parm2, parm3, parm4*****;**

wind_set() sets various window attributes.

OPCODE 105 (0x69)

AVAILABILITY All **AES** versions.

PARAMETERS *handle* specifies the window handle of the window to modify. *mode* specifies the attribute to change and the meanings of *parm1*, *parm2*, *parm3*, and *parm4* as follows:

Name	*mode*	Meaning
WF_NAME	2	This *mode* passes a pointer to a character string containing the new title of the window. *parm1* contains the high **WORD** of the pointer and *parm2* contains the low **WORD**.
WF_INFO	3	This *mode* passes a pointer to a character string containing the new information line of the window. *parm1* contains the high **WORD** of the pointer, *parm2* contains the low **WORD**.
WF_CURRXYWH	5	*parm1*, *parm2*, *parm3*, and *parm4* specify the x, y, w, and h of the new coordinates of the full extent of the window.
WF_HSLIDE	8	*parm1* specifies the new position of the horizontal slider between 1 and 1000. A value of 1 indicates that the slider is in its leftmost position.
WF_VSLIDE	9	*parm1* specifies the new position of the vertical slider between 1 and 1000. A value of 1 indicates that the slider is in its uppermost position.
WF_TOP	10	*parm1* specifies the window handle of the window to top. Note that if multiple calls of **wind_set(WF_TOP**, ... **)** are made without releasing control to the **AES** (which allows the window to actually be topped), only the most recent window specified will actually change position.
WF_NEWDESK	14	This *mode* specifies a pointer to an **OBJECT** tree which is redrawn automatically by the desktop as the background. *parm1* contains the high **WORD** of the pointer and *parm2* contains the low **WORD**. To reset the desktop background to the default, specify *parm1* and *parm2* as 0.
WF_HSLSIZE	15	*parm1* defines the size of the current slider relative to the size of the scroll bar as a value from 1 to 1000. A value of 1000 indicates that the slider is at its maximum size.
WF_VSLSIZE	16	*parm1* defines the size of the current slider relative to the size of the scroll bar as a value from 1 to 1000. A value of 1000 indicates that the slider is at its maximum size.

<table>
<tr><td>WF_COLOR</td><td>18</td><td>This mode sets the current color of the window widget specified on entry in parm1. Valid window widget indexes are as follows (W_SMALLER is only valid as of AES 4.1):

<table>
<tr><th>parm1</th><th>Value</th><th>ob_type</th></tr>
<tr><td>W_BOX</td><td>0</td><td>IBOX</td></tr>
<tr><td>W_TITLE</td><td>1</td><td>BOX</td></tr>
<tr><td>W_CLOSER</td><td>2</td><td>BOXCHAR</td></tr>
<tr><td>W_NAME</td><td>3</td><td>BOXTEXT</td></tr>
<tr><td>W_FULLER</td><td>4</td><td>BOXCHAR</td></tr>
<tr><td>W_INFO</td><td>5</td><td>BOXTEXT</td></tr>
<tr><td>W_DATA</td><td>6</td><td>IBOX</td></tr>
<tr><td>W_WORK</td><td>7</td><td>IBOX</td></tr>
<tr><td>W_SIZER</td><td>8</td><td>BOXCHAR</td></tr>
<tr><td>W_VBAR</td><td>9</td><td>BOX</td></tr>
<tr><td>W_UPARROW</td><td>10</td><td>BOXCHAR</td></tr>
<tr><td>W_DNARROW</td><td>11</td><td>BOXCHAR</td></tr>
<tr><td>W_VSLIDE</td><td>12</td><td>BOX</td></tr>
<tr><td>W_VELEV</td><td>13</td><td>BOX</td></tr>
<tr><td>W_HBAR</td><td>14</td><td>BOX</td></tr>
<tr><td>W_LFARROW</td><td>15</td><td>BOXCHAR</td></tr>
<tr><td>W_RTARROW</td><td>16</td><td>BOXCHAR</td></tr>
<tr><td>W_HSLIDE</td><td>17</td><td>BOX</td></tr>
<tr><td>W_HELEV</td><td>18</td><td>BOX</td></tr>
<tr><td>W_SMALLER</td><td>19</td><td>BOXCHAR</td></tr>
</table>

The ob_spec field of the object (containing the color information) while the window is on top is defined in parm2. The ob_spec field for the object while the window is not on top is defined in parm3.

This mode is only valid as of AES version 0x0300.</td></tr>
<tr><td>WF_DCOLOR</td><td>19</td><td>This mode sets the default color of newly created windows as with WF_COLOR above. This mode only works as of AES version 0x0300. As of AES version 4.1, this mode causes all currently displayed windows which have not had their color explicitly set with WF_COLOR to be changed.</td></tr>
<tr><td>WF_BEVENT</td><td>24</td><td>parm1, parm2, parm3, and parm4 are each interpreted as bit arrays whose bits indicate supported window features. Currently only one bit is supported. If bit 0 (B_UNTOPPABLE) of parm1 is set, the window will be set to be ‘un-toppable’ and it will never receive WM_TOPPED messages, only button clicks.

This mode is only available as of AES versions 4.0.</td></tr>
<tr><td>WF_BOTTOM</td><td>25</td><td>This mode will place the specified window at the bottom of the window list (if there is more than one window) and top the new window on the top of the list.

This mode is only available as of AES version 4.0.</td></tr>
</table>

WF_ICONIFY	26	This *mode* iconifies the specified window to the X, Y, width, and height coordinates given in *parm1*, *parm2*, *parm3,* and *parm4* respectively. Normally, this happens as the result of receiving a **WM_ICONIFY** message. This *mode* is only available as of **AES** version 4.1.
WF_UNICONIFY	27	This *mode* uniconifies the window specified, returning it to its original X, Y, width, and height as specified in *parm1*, *parm2*, *parm3*, and *parm4* respectively. Normally, this happens as the result of receiving a **WM_UNICONIFY** message. This *mode* is only available as of **AES** version 4.1.
WF_UNICONIFYXYWH	28	This mode sets the X, Y, width, and height that will be transmitted to the window with the next **WM_UNICONIFY** message that targets it. This call is used when a window is opened in an iconified state to give the OS a method of positioning it when it is uniconified. This *mode* is only available as of **AES** version 4.1.
WF_TOOLBAR	30	This *mode* attaches a toolbar to the specified window. *parm1* and *parm2* contain the high and low **WORD** of the address of the toolbar **OBJECT** tree respectively. *parm3* and *parm4* are unused. Set *parm1* and *parm2* to 0 to remove a toolbar.

BINDING

```
intin[0] = handle;
intin[1] = mode;
intin[2] = x;
intin[3] = y;
intin[4] = w;
intin[5] = h;

return crys_if(0x69);
```

RETURN VALUE **wind_set()** returns 0 if an error occurred or non-zero otherwise.

SEE ALSO **wind_get()**

wind_update()

WORD wind_update(***mode*** **)**
WORD ***mode*****;**

wind_update() manages the screen drawing semaphore.

OPCODE 107 (0x6B)

AVAILABILITY

All **AES** versions.

PARAMETERS

mode specifies an action as follows:

Name	*mode*	Meaning
END_UPDATE	0	This mode resets the flag set by **BEG_UPDATE** and should be called as soon as redrawing is complete. This will allow windows to be moved and menus to be dropped down again.
BEG_UPDATE	1	Calling this mode will suspend the process until no drop-down menus are showing and no other process is updating the screen. This will then set a flag which guarantees that the screen will not be updated and windows will not be moved until you reset it with **END_UPDATE**. Generally this call is made whenever a **WM_REDRAW** message is received to lock the screen semaphore while redrawing.
END_MCTRL	2	This mode releases control of the mouse to the **AES** and resumes mouse click message services.
BEG_MCTRL	3	This mode prevents mouse button messages from being sent to applications other than your own. **form_do()** makes this call to lock out screen functions. Desk accessories which display a dialog outside of a window must use this function to prevent button clicks from falling through to the desktop.

BINDING

```
intin[0] = mode;

return crys_if(0x6B);
```

RETURN VALUE

wind_update() returns 0 if an error occurred or non-zero otherwise.

VERSION NOTES

As of **AES** version 4.0, you may logically OR a mask of **NO_BLOCK** (0x0100) to either **BEG_UPDATE** or **BEG_MCTRL**. This mask will prevent the application from blocking if another application currently has control of the screen semaphore. Instead, if another application has control, the function will immediately return with an error value of 0.

This method should only be used by timing-sensitive applications such as terminal programs in which a long redraw by another application could cause a timeout.

COMMENTS

All **wind_update()** modes nest. For instance, to release the screen semaphore, the same number of **END_UPDATE** calls must be received as were **BEG_UPDATE** calls. It it recommended that you design your application in a manner that avoids nesting these calls.

Both the **BEG_UPDATE** and **BEG_MCTRL** modes should be used prior to displaying a form or popup to prevent them from being overwritten or clicks to them being sent to other applications.

Always wait until *after* the **BEG_UPDATE** call to turn off the mouse cursor when updating the screen to be sure you have gained control of the screen.

Applications such as slide-show viewers which require the whole screen area (and may need to change screen modes) may call **wind_update()** with parameters of both **BEG_UPDATE** and **BEG_MCTRL** to completely lock out the screen from other applications. The application would still be responsible for saving the screen area, manipulating video modes as necessary, restoring the screen when done, and returning control of the screen to other applications with **END_UPDATE** and **END_MCTRL**.

SEE ALSO **wind_new()**