Prospero C AES Bindings

Prospero
C
AES Bindings

September 1990

Prospero Software
LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

COPYRIGHT

DISCLAIMER

While every effort is made to ensure accuracy, Prospero Software cannot be held responsible for errors or omissions, and reserve the right to revise this document without notice.

TRADEMARKS

Acknowledgement is made for references in this manual to Microsoft and MS, which are trademarks of Microsoft Corp., to IBM, which is a trademark of International Business Machines Corp., to Apple and Macintosh, which are trademarks of Apple Computer Inc., to Digital Research and GEM, which are trademarks of Digital Research Inc., to Amstrad and Amstrad PC, which are trademarks of Amstrad Consumer Electronics plc, to Atari and Atari ST, which are trademarks of Atari Corp., to Motorola and MC68000, which are trademarks of Motorola Inc., and to Intel, which is a trademark of Intel Corp.

Prospero C, Pro Fortran-77, Prospero Fortran, Pro Pascal and Prospero Pascal are trademarks of Prospero Software.

Prospero Software, Inc.
100 Commercial Street, Suite 306
Portland, Maine 04101
U.S.A

Prospero Software Ltd.
190 Castelnau
London SW13 9DH
England

TABLE OF CONTENTS

1 INTRODUCTION TO GEM AES

GEM AES, Prospero C, and the Bindings

What is GEM AES? Why should you want to use it? Why do you need bindings before you can do so, and why should these bindings require such a large amount of explanation in order to be used?

GEM AES stands for Application Environment Services (Digital Research would have us believe that GEM also stands for Graphics Environment Manager, though as it was at one stage known as Crystal it seems likely that the name was invented before the acronym). The services which AES provides to an application are intended to allow applications to use the WIMP interface (this stands for windows, icons, mice and pull-down menus) which distinguishes GEM applications from other applications. The WIMP interface is a relatively new concept, designed to make computers and their software much easier to learn and use without having to remember (or look up in a manual) complicated commands such as `PIP lst:=b:mytext.txt[NT8]`. To anyone who is familiar with CP/M it might be immediately obvious that the above command should be used to cause a listing of the file mytext.txt on drive b to be printed, with line numbers and tabs expanded to 8 spaces, but someone who had never used CP/M would be hard pressed to guess what the above meant, let alone guess how to print a file by trial and error!

With a well designed program using a wimp interface, the user does not have to remember the commands to use, as they are set out as choices to be made in a series of pull-down menus. He or she does not need to remember what possible options can be chosen, as a dialog box will appear presenting all the choices that need to be made at any stage in the proceedings. A conventional program could prompt for each option in turn, but this will have the drawback of causing much of the previous screen contents to scroll away, and the user will have to answer questions about options about which they have no interest.

Ok, so you know what a wimp interface is – what makes GEM applications so different from any other applications which have well designed input screens, menus and so on? The first point is that every programmer has a different idea of exactly how the interface should be driven, and therefore the user has to learn a different set of keystrokes for every program that they use. Another problem is that since the interface is likely to use low level hardware features, implementing it on each new piece of hardware will involve a fair amount of work. GEM AES aims to solve both these problems by allowing all GEM applications to use a uniform interface, by means of a set of high level procedures which can be used on any hardware for which a GEM device driver has been written. Thus the writer of the application does not need to spend time and effort designing a menu handling routine, or working out how to support overlapping windows on screen – instead he or she can make the relevant GEM calls knowing that they will have the same effect whether the

program is going to run on an Atari ST, an Amstrad PC or any other computer using GEM. The user of the application does not need to spend time learning how to change the size of a window on their new GEM application, as it will be done in exactly the same way as on all the other GEM applications they are already familiar with. They will not need to remember what commands are available, as they can readily be summoned from the pull-down menus at the top of the screen.

The GEM AES and VDI are provided for each machine as programs which are resident in memory at the same time as the GEM application which uses their facilities, and are used by setting up values in areas of memory, whose addresses are then placed in a register and a software interrupt generated to cause entry to the GEM code. If an application is being written in assembly language, then that would be how the application would access the GEM routines, but in order to access them from a high level language such as Prospero C, a better method needs to be provided. For each of the routines provided by the GEM AES, a corresponding function is provided, callable from the high level language, which sets up the required values in the right areas of memory and makes the software interrupt, then copies any return values out of the memory areas back into variables of the high level language. These functions are known as the bindings. When Digital Research designed the GEM procedures, they also designed a set of C bindings. The Prospero C bindings are based on these original C bindings, but have been extended, and rewritten to use the function prototypes introduced into the C language by the proposed ANSI C standard.

Most of the time the Prospero C bindings are completely compatible with the original Digital Research C bindings, and programs designed to use the standard bindings can be compiled and linked with the Prospero bindings without changes.

The header file AESBIND.H defines a number of types and macros, and most of these are described in the section to which they apply. However, the following types occur frequently throughout the manual :-

```
typedef short int WORD;     /* All integer values */
typedef WORD *LONG;         /* General pointer to WORD array */
```

The low level interface to GEM AES need not usually concern the C programmer – all such details are handled by the bindings as transparently as possible. However the following may be of interest in some circumstances – further information can be found in the GEM Programmer's Toolkit, available from Digital Research.

GEM AES is entered by a software interrupt, with the address of a parameter block contained in a register. This parameter block contains the addresses of 6 arrays through which information is passed to and from GEM AES, and is declared as an `extern struct` as follows :-

```
extern struct AESparmblock {
    WORD *pcontrol, *pglobal, *pintin, *pintout;
    LONG *paddrin, *paddrout;
   } AESparm;
```

These pointers are initialized to the addresses of six arrays, which are declared as follows :-

```
extern WORD AES_control[5];
extern WORD AES_global [15];
extern WORD AES_intin  [17];
extern WORD AES_intout [8];
extern LONG AES_addrin [3];
extern LONG AES_addrout[2];
```

The above declarations are included in the file AESBIND.H so that they can be accessed directly from your program should you so wish – this will not normally be necessary.

The `AES_control` array is used for each GEM call to indicate which GEM AES routine is required, and how many parameters are being passed in the other arrays as follows :-

Element	Purpose
0	the function number required
1	the number of values passed in the `AES_intin` array
2	the number of values returned in the `AES_intout` array
3	the number of values passed in the `AES_addrin` array
4	the number of values returned in the `AES_addrout` array

The `AES_global` array contains information about the application, set up by the call of `appl_init` at the start of the application, and occasionally referred to in subsequent GEM AES routines. The information it contains is seldom of direct interest to an application, and can generally be obtained by means of various GEM functions. For details, see the GEM Programmer's Toolkit.

The AES_intin array contains two-byte values to be used by the AES. In general, all coordinates, numbers, flags, characters and so on passed as values to the bindings are copied into the required position in this array. Note that (unlike the VDI), when strings are passed in the AES, the address of the string is passed via the AES_addrin array rather than by copying the characters into the AES_intin array, so that none of the AES bindings ever make use of more than 16 words in this array.

The AES_intout array is used to return two-byte values from AES functions, which are then copied into the objects pointed to by any relevant parameters by the bindings when values are to be returned. The first element of the array always contains a copy of the function result.

The AES_addrin and AES_addrout arrays are used to pass 32-bit addresses to and from GEM AES – these might for example be the address of a tree structure (see section 6) or of a message buffer.

2 USING GEM AES

GEM AES contains a large number of routines which, once you understand them, allow you to create a GEM application complete with all the standard features of the GEM wimp interface. Before describing the functions in detail, it seems in order to describe just what those features are, how a typical GEM application works, and why it works in that way.

The features to be found in a typical GEM application are as follows :-

1. It uses a mouse, which can be moved around on any flat surface and causes a corresponding cursor form to move around the screen. This mouse is used for many purposes – it can be used to indicate selection of a particular item, by pressing or releasing the button when the cursor is over an object on the screen which represents that item; it can be used to select a group of items by marking an area, pressing the mouse button at one corner, moving the pointer to the opposite corner and releasing the button; it can be used to request a particular action on a window by clicking or dragging one of a number of window control points. Applications can make the mouse serve other purposes as well, but in almost every GEM application it will serve the above purposes.

2. It has a pull-down menu bar across the top of the screen. When the mouse cursor is moved within this area, a menu drops down with a list of options to select. An option is selected by moving the mouse pointer over the option and clicking.

3. It uses overlapping windows, which are used to separate different classes of output. These windows can normally be manipulated by the user by clicking the mouse button when the pointer is over particular portions of the window. Thus the user can change the size and position of each window, control which window is currently on top, and so on.

4. It uses dialog boxes to get information from the user. These are drawn over the windows giving a number of options to be selected by clicking in boxes, and/or text to be filled in. The appearance of the dialog box changes to reflect the choices made. When the user has finished making choices, they click in a box (usually marked OK) and the dialog box is removed from the display by redrawing whatever was obscured by it. The dialog boxes behave to some extent like paper forms to be filled in – the user is free to fill them in in whatever order they like, to ignore some sections and so on.

So how does an application provide all the above features? A lot of the work involved in providing such an interface is performed by GEM AES, so that it is relatively simple to provide a sophisticated, user friendly interface that will immediately be familiar to users as being consistent with all other GEM applications. The place to start is with a resource file.

The resource file is a concept that originated on the Apple Macintosh as a means of allowing programs to be easily convertible to different countries, so that all messages could be translated without having to recompile the program. The concept rapidly progressed, so that resources on the Macintosh are now used to contain a huge variety of information that may be of use to an application. Some of this idea has been used in GEM, so that data structures to be used by an application can be created once and for all in the resource file then loaded into memory to be used by the application. This process is very much preferable to having to construct the data structure dynamically every time the program is run – this is expensive in coding effort, code space and execution time.

The resource file of a typical GEM AES application will contain a menu bar, several dialog forms, perhaps some icon definitions, perhaps some strings to be used for alerts etc. It is a good idea to place all messages in the resource file for translation to different countries. The resource file is created using a resource editor (for example the RCS provided with the GEM Programmers Toolkit, or one of the resource editors available separately), which also creates a header file to be included by a C program, defining macros to give values to names by which the program can refer to the various entities within the resource file. The application loads this file into memory using `rsrc_load` (section 12.1) as part of its initialization process, and can then obtain the addresses of any objects within it using `rsrc_gaddr` (section 12.3)

For simple applications, or where it is important to have a single file rather than separate resource and code files, Prospero C provides several extra functions for dynamically creating menu bars and dialog forms. These are described in section 5.8 to 5.10, and 6.16 to 6.18. Note however that the use of these functions is likely to increase the code size by more than the size of the equivalent resource file, so that the use of resource editor is recommended where practical.

The first stage of an application's initialization process is to call `appl_init` (section 3.1), which allows GEM AES to allocate any space it needs to store information about the current state of this application, and generally prepare itself for use. The next step is to load the resource file using `rsrc_load`, or dynamically create the menu bar and any dialogs which will be used, as described above. If the resource file cannot be found, the application will not be able to proceed any further – it can either invite the user to indicate the directory and name where the resource file is to be found, or more normally

simply report that the resource file is missing using `form_alert` (section 7.3) then terminate.

The next stage of the initialization is to prepare the menu bar for use – the address within the resource file is obtained using `rsrc_gaddr`. Often an application will want to set up the state of some menu items to reflect the current setting of any options – for example any menu items which are not immediately appropriate must be disabled using `menu_ienable` (section 5.3), those which correspond to currently selected options might be checked using `menu_icheck` (section 5.2) or have their text altered using `menu_text` (section 5.5). Once the menu bar is suitably initialized, it can be displayed and enabled using `menu_bar` (section 5.1).

The application may also wish to open some windows at this stage – perhaps to represent a new empty document for a word processor or editor application, or to contain the text of the document which was opened to start the application (this can be discovered using `shel_read` – section 13.1). To open a window, it is necessary to discover the resolution of the screen – more specifically, the coordinates and size of the area of the screen below the menu bar is required. This area is usually referred to as the Desktop, and is regarded as a window with the special window handle zero. Thus the application can use `wind_get` (section 11.5) to obtain the size of its work area. The size of this work area will normally be used as a window's maximum size when creating a window using `wind_create` (section 11.1) – certainly the window must not be larger. Creating a window in this way does not cause it to be displayed on screen – this is achieved using `wind_open` (section 11.2). The application should the determine the work area of the newly opened window using `wind_get` (section 11.5) and is then free to output VDI graphics or text to this area, or to draw an AES object tree within it, as described in section 6.

Once all the initialization is complete, the application is ready to accept user input. The user may respond in a number of ways – for example by typing keys, by selecting a menu item, by clicking the mouse button, by moving the mouse, or simply by doing nothing for a while. Each of the above is an example of what is known in GEM AES as an event, and can be detected using one of the functions in the event library (section 4). GEM applications normally operate by waiting for an event, performing whatever action is indicated by that event, then waiting for the next event. Thus each application will have a main event loop, which simply waits for each event and calls the relevant processing routine. The heart of this routine will almost always be a call to `evnt_multi` (section 4.6) which allows the detection of any of a number of different occurrences. A simple program, or one in the early stages of development, may use a call to `evnt_mesag` (section 4.4) instead – this will allow the detection of message events only, and ignore any keystrokes. As menu selections and manipulation of window control points are both notified using messages, quite a lot can be achieved without the need for any keyboard

input. Forms and dialogs (described in sections 6 and 7) can be used to request further information on what is required from the user.

If any windows are displayed on the screen, then unless the application is certain that all parts of every window will be visible at all times, the application will need to be able to respond to redraw messages from the screen handler when portions of the windows are revealed. It will be extremely unusual for this not to be necessary, as the use of a menu bar implies that desk accessories can be started up (and can therefore obscure any portion of the screen). The use of any dialog boxes (other than alert boxes) will also cause areas of the screen to be obscured. An application will also need to respond to other window control messages if any windows displayed use any of the features (other than NAME or INFO) described in section 11.

To respond to a window control message is usually fairly straightforward – for example to respond to a WM_MOVED message the application would simply set the window's coordinates using `wind_set` (section 11.6) to those requested in the message. You might therefore wonder why the message is sent at all – why could GEM AES not simply move the window ? The reason for sending a message is so that the application can take other action – for example the application might allow windows to lie only in certain positions.

To respond to a redraw message is slightly more complicated – the application must output the window's contents, but only to those portions of the window which are visible. The procedure for doing this is described in more detail in section 11.

Having repeatedly waited for an event, processed it, then waited for another event, the application will eventually be asked to process the event that corresponds to the user selecting QUIT from the menu. At this point the application should tidy up and then end, to return to the GEM Desktop application. The tidying up is extremely important, as if it is performed wrongly the next application to execute may find itself in trouble.

The first step in the tidying up is to close and delete all windows. Normally an application will know whether any window represents unsaved work by the user, and invite the user to save it before quitting. The window may then be closed using `wind_close` (section 11.3), which causes it to be removed from the screen, then deleted using `wind_delete` (section 11.4) to allow the window handle and its associated workspace to be reused by a subsequent application.

Once the windows are cleaned up, all that remains is to disable the menu bar using `menu_bar` (section 5.1) and release the resource file memory using `rsrc_free` (section 12.2). The application then calls `appl_exit` (section 3.1) to indicate that it no longer requires the use of any of GEM AES's functions or data structures, and terminates.

Desk accessories are very similar, the main difference being that they do not terminate, but should execute the main event loop indefinitely. Desk accessories cannot easily be debugged, so it is usually best to write the accessory as a normal application at first, then only when it is working convert it into a desk accessory. This requires a few small changes to the program logic, as described below.

Desk accessories should not initially create or open any windows, as these would not then be available to the main application. In fact very little initialisation is required other than calling `menu_register` (see section 5.6) to install the accessory in the Desk menu. The accessory can then wait for an AC_OPEN message indicating that the menu item has been selected, and the desk accessory is to become active. At this point, the accessory should open any windows it requires, and enter its main event loop, just as for a normal application. If another AC_OPEN message is received while the accessory is still active, it should bring its window to the top rather than opening a new one, to avoid running out of windows. When an AC_CLOSE message is received, or a WM_CLOSED message if only one window is open, the accessory should close and dispose all windows, release any other memory it does not require, and return to the inactive state, waiting for an AC_OPEN message.

3 APPLICATION LIBRARY

This section contains descriptions of the Application Library functions, in the following sub-sections.

Section	Function description	Binding name
3.1	Initialize Application	`appl_init`
3.2	Pipe Read and Write	`appl_read` `appl_write`
3.3	Find Application	`appl_find`
3.4	Record and Playback Events	`appl_trecord` `appl_tplay`
3.5	Application Disk Set	`appl_bvset`
3.6	Application Yield	`appl_yield`
3.7	Application Exit	`appl_exit`

The functions in the Application library are assorted housekeeping routines, concerned with controlling the initialization and use of various application data structures.

3.1 Initialize Application `appl_init`

Initialize Application is provided to alert GEM to the fact that you are going to run a GEM application and that it should provide you with various facilities, and initialize global information about your application.

3.1.1 Definition

The Prospero C definition of Initialize Application is :

```
WORD appl_init(void);
```

3.1.2 Purpose

An application should call this once at the start of any program which makes calls to GEM AES. It sets up various global areas that are used by all other AES functions.

3.1.3 Parameters

There are no parameters.

3.1.4 Function Result

The value returned is the application's global identifier, which may be needed for one or two other GEM AES function calls. If this value is –1, the application could not be initialized, and should terminate without making any further GEM AES calls.

3.1.5 Example

```
if (appl_init() == -1)
  exit(3);              /* Unsuccessful initialization */
```

3.2 Pipe Read and Write

appl_read
appl_write

The application pipe is a message buffer which is specific to each application; an application will normally read from its own pipe and write to other applications' pipes. The commonest use of pipes is for messages sent by the GEM screen handler to indicate menu selection or window manipulation, in which case messages will be received using the `evnt_mesag` or `evnt_multi` functions described in section 4. However, an application can use the message pipes for other, more general communication between itself and other applications or desk accessories, using these two functions.

3.2.1 Definition

The Prospero C definitions of Pipe Read and Pipe Write are :

```
WORD appl_read (WORD id, WORD length, WORD pbuff[]);
WORD appl_write(WORD id, WORD length, WORD pbuff[]);
```

3.2.2 Purpose

These functions allow an application to read a message from its own pipe or to write to its message pipe or that of another application. A common use of `appl_write` is for an application to send a message to itself to simulate for example a menu selection or a GEM redraw request – it can also be used to send a message to a desk accessory such as a print spooler.

It is not normally necessary to use `appl_read` to receive messages, as the standard 16-byte messages from the GEM screen handler are received using `evnt_mesag` or `evnt_multi` (see section 4); however, if a message is received (via `evnt_mesag`) with a non-zero value in word 2, indicating that the message length was not 16 bytes and there is more to come, `appl_read` should be used to read the remainder.

There is a standard message format for passing information between applications, using a 16-byte message to pass the address of a larger message as follows :-

word 0 : Message type (1024 to 32000)
word 1 : Sender's application identifier
word 2 : –1
word 3 : Length of message at the given address
word 4 : First word of address of message
word 5 : Second word of address of message
word 6 : Application specific
word 7 : Application specific

If the above message passing protocol is in use, `appl_read` will not be used, as the 16-byte message described above should be received via `evnt_mesag` or `evnt_multi`.

3.2.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`id`	*WORD*	**Identity number** The application identifier of the application whose pipe is to be read or written. This can be obtained from `appl_init` or `appl_find`.
`length`	*WORD*	**Length** The number of bytes to be read or written.
`pbuff`	*WORD[]*	**Pipe buffer** The parameter `pbuff` gives the location to or from which the message is to be transferred. A pointer to an array of suitable size should be passed. It is up to the application to ensure that the array is large enough to hold the number of bytes being read or written.

3.2.4 Function Result

The value returned is zero if an error occurred, or greater than zero if no error was detected.

3.2.5 Example

```
WORD pipe_buffer[20];
WORD my_id;

main()
{ my_id = appl_init();
  .
  .
  /* Send myself a message */
  appl_write(my_id, 40, pipe_buffer);
                              /*40 bytes = 20 words*/
```

3.3 Find Application `appl_find`

In order to write to the message pipe of another application, an application needs to find its `ap_id` number; this function does that.

3.3.1 Definition

The Prospero C definition of Find Application is :

```
WORD appl_find(char pname[9]);
```

3.3.2 Purpose

This is used to discover the application identifier of another active application.

3.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`pname`	*char[9]*	**Application name** This parameter is the filename of the application whose identifier is to be returned. The filename passed should be the (up to) 8 characters before the dot separating the name from the extension, terminated with a null character.

3.3.4 Function Result

The value returned is a *WORD* containing the application identifier of the specified application, if it is active. A value of –1 indicates that the application is not active.

3.3.5 Example

```
WORD cal_id;
WORD message_buffer[8];
.
.
  cal_id = appl_find("CALCLOCK");
  .
  /* Send a message to the calculator accessory */
  appl_write(cal_id, 16, message_buffer);
```

3.4 Record and Playback Events `appl_trecord` `appl_tplay`

This pair of functions allow an application to record and playback a series of mouse or keyboard events. They could be used to implement teaching programs, demonstration programs, a macro facility, or to provide a repeat-last-action facility, or an undo-last-action facility (though this is likely to be more complicated since any data thrown away has to be recorded in case it is needed for undo).

3.4.1 Definition

The Prospero C definitions of Record and Playback Events are :

```
WORD appl_trecord(WORD tbuffer[], WORD tlength);
WORD appl_tplay(WORD tbuffer[], WORD tlength,
                WORD tscale);
```

3.4.2 Purpose

These two functions allow the application to record or play back a series of mouse or keyboard events. The events are stored in an array of structures of the following form :-

8086 processors :

```
struct {
     WORD event; /* 0,1,2,3 => timer, button,
                               mouse, keybd */
     long info;
    }
```

68000 processors :

```
struct {
     long event; /* 0,1,2,3 => timer, button,
                               mouse, keybd */
     long info;
    }
```

The meaning of the info field depends upon the event in question, but is always 4 bytes in length. For a timer event, it gives the number of milliseconds elapsed; for a button event, the low word is the button state and the high word is the number of clicks; for a mouse event the low and high words contain the mouse's X and Y coordinates respectively, while for a keyboard event the low word contains the character typed, the high word contains the keyboard state. More information on all these values can be found under the relevant routine in the event library.

3.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
tbuffer	*WORD[]*	**Recording buffer** This parameter is used to pass the address of the array which stores the events. This should be an array of tlength objects of the form described above.
tlength	*WORD*	**Recording length** This parameter gives the number of events to be recorded or played back. The application must ensure that this does not exceed the size of the recording array.
tscale	*WORD*	**Playback speed** This parameter controls playback speed, from 1 to 10000, where 100 is normal speed, 50 is half speed, 200 is double speed and so on. (appl_tplay only)

3.4.4 Function Result

The function `appl_trecord` returns the number of events recorded – normally this will be the same as the value passed in the parameter `tlength`, but in GEM version 2.0 it is possible to terminate a recording by entering ctrl-backslash (\).

The function `appl_tplay` always returns 1.

3.4.5 Example

```
long buffer [400];
WORD      i, size;
FILE *evntfile = fopen("evntdata", "wb");
.
.
  size = appl_trecord(buffer, 400);
  /* Write them to a file */
  fwrite(evntfile, buffer, size, sizeof(long));
  .
  .
  /* Play them back full speed */
  appl_tplay(buffer, size, 100);
```

3.5 Set Disk Configuration `appl_bvset`

This function allows an application to set the disk configuration information used by GEM. It is not provided in GEM version 1.1

3.5.1 Definition

The Prospero C definition of Set Disk Configuration is :

```
void appl_bvset(WORD bvdisk, WORD bvhard);
```

3.5.2 Purpose

This is used to tell GEM what disk drives are in the system, and which are hard disks.

3.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
bvdisk	*WORD*	**Disk bit vector**
bvhard	*WORD*	**Hard disk bit vector**
		These parameters are interpreted as 16 one-bit values indicating which drive letters A to P correspond respectively to disk drives and hard disk drives in the system. The most significant bit represents drive A, with a bit value of 1 indicating that the drive is present.

3.5.4 Function Result

There is no function result.

3.5.5 Example

```
/* Drives A, B, C; C is hard */
appl_bvset(0xe000, 0x2000);
```

3.6 Application Yield `appl_yield`

Application Yield is provided so an application can allow other active applications or desk accessories to execute. Normally GEM will check whether it is another application's turn to run whenever an application makes an event library call. If an application does not make any event calls for a while, it will not allow other applications or accessories to operate during this time. This is not desirable, especially in Multitasking GEM which may appear shortly – if an application is not making event library calls (perhaps during a large computation, or a compilation) it should call this function every (say) second or so to allow other applications or accessories to have their events processed.

This function is not supported by GEM version 1.1, though a similar effect can be achieved using `evnt_timer` (see section 4.5) with a delay of zero.

3.6.1 Definition

The Prospero C definition of Application Yield is :

```
void appl_yield(void);
```

3.6.2 Purpose

This function is used by an application to force a dispatch, to allow other applications' events to be processed if any are ready.

3.6.3 Parameters

There are no parameters.

3.6.4 Function Result

There is no function result.

3.6.5 Example

```
long int i;

for (i = 0; i < 100; i++)
{  calculate(i);
   appl_yield();
   /* yield every so often in a big calculation */
}
```

3.7 Exit Application `appl_exit`

Exit Application is provided to alert GEM to the fact that an application is about to terminate, and no longer requires various facilities.

3.7.1 Definition

The Prospero C definition of Exit Application is :

```
WORD appl_exit(void);
```

3.7.2 Purpose

This function is used by an application to notify GEM AES that no further AES calls are to be made, and is normally called by a program which is about to terminate. Once this call has been issued, GEM will reallocate the memory previously occupied by an application's global information, and reuse the application's identifier. No GEM calls should be issued after `appl_exit`. It is important to tidy up windows and so on before terminating – all windows should be closed and deleted (in that order), and `wind_update` (see section 11.8) must have been correctly used so that the application is not in window update or mouse control mode, otherwise the next application (usually the GEM desktop) will not be able to run correctly.

3.7.3 Parameters

There are no parameters.

3.7.4 Function Result

The value returned is zero if an error occurred, or greater than zero if no error was detected.

3.7.5 Example

```
main()
{ if (appl_init() != -1)
  { /* Do lots of GEM calls */
    ...
    appl_exit();
  }
}
```

4 EVENT LIBRARY

This section contains descriptions of the Event Library functions, in the following sub-sections :-

Section	Function description	Binding name
4.1	Wait For Keyboard Event	`evnt_keybd`
4.2	Wait For Button Event	`evnt_button`
4.3	Wait For Mouse Event	`evnt_mouse`
4.4	Wait For Message Event	`evnt_mesag`
4.5	Wait For Timer Event	`evnt_timer`
4.6	Wait For Multiple Events	`evnt_multi`
4.7	Set Double Click Delay	`evnt_dclick`

GEM applications work in an event driven manner: rather than continually checking the keyboard, the mouse, the message pipe, the menu bar and so on to see if anything has happened, they decide what they want to happen, then wait until it does. A typical GEM application will have one loop in which it waits for an event, usually one of several, so that the user can respond by pressing a key, clicking the mouse, or selecting a menu item or a window control point. When an event occurs, the application will determine which sort of event it was, and what the user intended its effect to be, then go away and cause that effect to happen. When the processing generated by that event is over, the application returns to the main loop to wait for the user's next instruction. These events should not be confused with interrupts, as GEM AES has no way of notifying an application that something has happened until the application asks. GEM events are basically an efficient way of handling all input from the user to an application. Note that the GEM VDI input functions should not be used by an application which uses the AES – all input from the user should be by means of the event library, or using the form library to allow the user to interact with a dialog or alert box (this will itself make use of the event library routines internally).

GEM is in a very limited way a multi-tasking system. Large computers have complex operating systems which share the processor power among a number of users so that each task appears to the user to be running in a separate computer. GEM does this to a small extent, in that desk accessories can continue to operate while another program is active. The way in which this is achieved is by using events to receive input – when an application is waiting for input, it calls one of these functions, which notifies GEM AES that the application can do no more until the specified event occurs. GEM AES will check whether any active application was waiting for an event which has now occurred, and when it finds one that application will be brought into context. In standard GEM, only one main application can be active at any time, but a number of desk accessories can also be active and waiting for events – in fact all desk accessories installed will be waiting for an event of some sort – if they haven't yet been opened they will be waiting for a message event which GEM AES will send to them when the user selects their menu item. If each accessory in the system was given a share of the processor to check every so often whether a message had occurred, there would be very little processor time left. By using events, GEM AES can see what has happened then check who might be interested, which is a very much more efficient way to behave.

There are 5 different types of event corresponding to different user actions or forms of input, as follows :-

Keyboard events	– a key is pressed
Button events	– the state of the mouse buttons changes
Mouse events	– the user moves the mouse into or out of an area of the screen
Message events	– these can come from a variety of sources, and are described further later
Timer events	– the user does nothing for a while

An application can wait for each of these types of event, or it can wait for any one of a combination of them, depending upon what input is appropriate. Waiting for a single event is not particularly useful, as the application would not then be notified of events of any other type. Even if a particular input is not appropriate, it is more user-friendly if an application responds in some way to that action, perhaps by beeping if the mouse is clicked in the wrong place, or if a key is pressed when no keyboard input is appropriate.

Keyboard Events

Keyboard events occur when a key is pressed on the keyboard. The application can discover what the key was, as both its ASCII code and scan code are returned. GEM treats all keyboards as a standard keyboard – this is very useful when trying to write portable applications, as the ASCII and scan codes will always relate to the same keys. When it receives a keyboard event, the application can discover if it wishes whether the control, alt or shift keys are depressed, using the function `graf_mkstate` (section 8.9).

Button Events

Button events occur when the state of a particular mouse button or buttons enters (or in GEM version 2.0 leaves) the state for which the application is waiting. The application can specify which buttons it is interested in, and what state it is waiting for. In order to recognise double clicks in a consistent way in all applications, and at the same speed, double click detection is built in to button event detection. The application specifies the maximum number of times that the specified button state is to be detected before the event is reported. After detecting the mouse button state, the event manager will not return immediately, but continue to check the mouse button state until either the specified maximum number of clicks has been detected, or a standard time delay has passed, returning the number of clicks that were counted. In this way an application can specify whether it wants double clicks to be recognised, and detect when they occur.

Mouse Events

Mouse events occur when the mouse enters or leaves an area of the screen. They are normally used to change the mouse form depending upon what part of the screen it is over, and in particular to fulfil GEM's requirement that mouse forms other than the arrow or busy cursor may only be used within the work area of the active window. Mouse events are also useful for detecting movement of the mouse, for example when 'dragging' by holding down the button while moving the mouse. If an application wished to perform a different action depending upon whether the mouse is clicked or dragged, it would first wait for a button event indicating that the button was down, then wait for one of the following events to occur: either a button event indicating that the button has been released, or a mouse event indicating that the mouse has left the area of the screen that it previously occupied. The size of the rectangle specified depends upon the amount of movement permitted before the application wishes to recognise a drag, and could be anything from one pixel square upwards.

Having detected a drag, the application will continue to wait for the same two events, changing the screen to reflect the effect of the drag whenever the mouse moves, until it detects the button has been released. The function `graf_dragbox` (section 8.2) may be used in many situations to do much of this work for the application.

Timer Events

Timer events occur when a specified amount of time has elapsed. On their own they are not very useful, but they can be used to put a limit on the wait for other events, perhaps to check periodically for a button press while continuing to calculate rather than suspending all calculations until the button is pressed.

Message Events

Message events are perhaps the most important sort of events, which make all the difference between GEM applications and non-GEM applications. Messages can be sent from other processes, such as desk accessories, but this is not their most important function. Messages are also sent to applications by the GEM AES screen manager, which is responsible for monitoring all interaction between the mouse and the menu bar and window control points like the slider bars, size box and so on. When the user makes a menu selection, or clicks on or drags a window control point, the application must know that this has happened, and which item or control point was selected, and what was done with it. If each application had to detect a mouse click, check which window control point it was over, monitor the mouse as it dragged a slider, and so on, this would make all applications very much larger, and almost certainly make all their interfaces different, Instead of this, all such interactions are handled by the AES screen manager. Only when the user has decided where to place the slider and released the button does GEM AES notify the application what has happened, and where the slider has been placed. It does this by sending a message, in one of the standard predefined forms. All message events return an 8-word message in the following format:

word 0	– the type of the message (see below)
word 1	– the identifier of the process that sent the message
word 2	– the length of the message, excluding the 8 words here
words 3 to 7	– meaning depends upon message type

All messages from the screen manager are exactly 8 words, so contain zero in word 2. However, messages passed between applications can define their own message types and forms, so long as the first 3 words correspond to this convention. Any additional portion of the message (if this value is not zero) must be read using `appl_read` (section 3.3).

The predefined messages sent by the screen manager are as follows:

Word 0 Name (in AESBIND.H)

10 MN_SELECTED

Sent when the user selects a menu item. The object indices of the title and item are returned in words 3 and 4 of the message respectively.

20 WM_REDRAW

Sent to an application when part of the work area of one of its windows needs redrawing. The handle of the window is given in word 3 of the message, while words 4 to 7 give (in raster coordinates) the x and y coordinates, width and height of the area to be redrawn. This should be intersected with the window's visible rectangle list before redrawing – see section 11 for more information.

21 WM_TOPPED

Sent when the user has requested that a new window is made active. The handle of the window concerned is passed in word 3 of the message. The application can carry out this request by making the specified window active using `wind_set` (section 11.6).

22 WM_CLOSED

Sent when the user has clicked in the close box of the active window. The window's handle is passed in word 3 of the message. The application can carry out this request by using `wind_close` (section 11.3). Note that the screen manager does not itself close the window when the user clicks in the close box, but asks the application to do it. This allows the application to reject the request if, for example, the window represents a document that has not yet been saved.

23 WM_FULLED

Sent when the user clicks in the active window's full box, in the top right of the title bar. The window's handle is passed in word 3 of the message. By convention, GEM applications respond to this by enlarging the window to its maximum size, or shrinking it to its previous size if it is already full size. This can be done using `wind_set` (section 11.6).

Word 0 Name (in AESBIND.H)

24 WM_ARROWED

Sent when the user clicks on an arrow, indicating a scroll of one 'row' or 'column', or in the slider bar, indicating a scroll of one page. The handle of the active window is given in word 3 of the message, while word 4 contains a code indicating what type of scroll has been requested, as follows:

0 page up
1 page down
2 row up
3 row down
4 page left
5 page right
6 column left
7 column right

The application is free to decide how to interpret a row, column or page, or to ignore the request. If the application does redraw the contents of the window to reflect the scroll, it should also alter the slider position to reflect the change, using `wind_set` (section 11.6).

25 WM_HSLID

Sent when the user has dragged the horizontal slider to a new position. The handle of the active window is given in word 3 of the message. Word 4 contains a value in the range 1 to 1000, indicating the slider position requested by the user, 1 meaning the left-most position, 1000 meaning the right-most. The application will normally redraw the window to display the requested portion of the document or image, and alter the position of the slider to reflect the change. The new slider position set by the application need not be the same as that requested – normally the closest value that corresponds to an exact 'column' of whatever the window contains will be used.

26 WM_VSLID

Sent when the user has dragged the vertical slider to a new position. The handle of the active window is given in word 3 of the message. Word 4 contains a value in the range 1 to 1000, indicating the slider position requested by the user, 1 meaning the top and 1000 meaning the bottom.

Word 0 Name (in AESBIND.H)

27 WM_SIZED

Sent when the user drags the active window's size box to a new position, indicating that the window's size should be altered. The handle of the window is passed in word 3 of the message, while words 4 to 7 contain the requested coordinates, width and height of the window (the coordinates will be the same as those already set for the window). The application will normally select the nearest appropriate size to that requested, and set the window to that size using `wind_set` (section 11.6). The application should not respond to this message by redrawing any new portions of the window – if the window is enlarged in either direction, GEM AES will send a redraw message to make the application redraw the new portion anyway.

28 WM_MOVED

Sent when the user drags the active window to a new position by dragging in the title bar. The handle of the window is passed in word 3 of the message, while words 4 to 7 contain the requested coordinates, width and height of the window (the width and height will be the same as those already set for the window). The application will normally select the nearest appropriate position to that requested, and set the window to that position using `wind_set` (section 11.6). If the window was previously partially offscreen, this may cause a redraw message to be issued. Note that if an application is outputting text to a window, the process will be very much faster if the text is output to a word-aligned pixel. Many applications therefore select the nearest window position to that requested which will cause the text displayed within to be on a word boundary.

30 WM_UNTOPPED

Sent when an application's window is about to be made inactive, and therefore liable to be obscured. The handle of the window is passed in word 3 of the message. The application can then save any image contained in the window, or take other appropriate action. This message is not sent in GEM version 1.1.

Word 0 Name (in AESBIND.H)

40 AC_OPEN

Sent to a desk accessory application when its menu item has been selected. Word 3 of the message contains the menu item identifier of the accessory selected, as returned by `menu_register` when the name was placed in the desk menu. An application which contained two or more accessory functions, and therefore added more than one name to the Desk menu, would use this value to see which had been requested. The accessory application should open a window and activate the required accessory, or if the window is already open, it should bring it to the top.

41 AC_CLOSE

Sent to a desk accessory application when the current application terminates, or the screen is about to be cleared. The accessory should close and delete any windows, then await an AC_OPEN message. Word 3 of the message contains the accessory's menu item identifier.

Many of the above messages correspond closely to the window control points described in section 11. It is important to note that a window which does not have a particular feature cannot cause the message corresponding to that feature to be generated. Note also that when a user interacts with the window, by dragging it or clicking its boxes, this does not in itself cause the window to be moved or whatever. It is purely by convention that when an application receives a WM_MOVED message, it moves the window to the specified position. A perverse programmer could easily create a window whose close box caused the window to be fulled, and whose full box caused the window to be closed. Such an application is unlikely to be popular with its users – one of the best features of GEM applications is that they have a uniform interface, so that the amount a user has to learn and remember is greatly reduced.

4.1 Wait For Keyboard Event `evnt_keybd`

Wait For Keyboard Event is the basic get character function of GEM; it can also be used to get any other key. This function should only be used where a character on the keyboard is the only possible response the user can make. It will not be widely used except in simple programs, as a GEM application normally offers the user several alternative means of response. This is possible using `evnt_multi` (section 4.6)

4.1.1 Definition

The Prospero C definition of Wait For Keyboard Event is :

```
WORD evnt_keybd(void);
```

4.1.2 Purpose

This function is used to wait for any key to be pressed.

4.1.3 Parameters

There are no parameters.

4.1.4 Function Result

The value returned is a 2-byte integer, whose low byte gives the ASCII code of the key pressed (0 for a non-ASCII key, such as a function key) and whose high byte gives the standard IBM PC scan code of the key. These can be referenced as `result%256` and `result/256` respectively.

4.1.5 Example

```
char ch;

/* Wait for a capital letter */
do {
     ch = evnt_keybd() % 256;
   } while (!isupper(ch));
```

or simply

```
  evnt_keybd();  /* Wait for a key before continuing */
```

4.2 Wait For Button Event `evnt_button`

Wait For Button Event is used to wait until a particular mouse button state is detected. This might be used when processing a simple form, or at other times when no other events are allowed. If the application wishes to recognise other events as well, such as keys being typed or menus being selected, it must use `evnt_multi` (section 4.6).

4.2.1 Definition

The Prospero C definition of Wait For Button Event is :

```
WORD evnt_button(WORD clicks, WORD mask, WORD state,
                 WORD *pmx, WORD *pmy,
                 WORD *pmb, WORD *pks);
```

4.2.2 Purpose

This function waits until the state of the mouse buttons matches that specified by the parameters `mask` and `state`. The function will return when the required state has been entered the number of times specified by the `clicks` parameter, or when a certain time interval has passed since the state was detected for the first time – this delay can be adjusted using the function `evnt_dclick` (section 4.7). Thus if the value of `clicks` is 1, the function will always return as soon as the required state is detected, while if it is greater than 1, the function will return shortly after the state was first detected, counting how many times the state is reentered up to the value specified. An application should pass a value of 2 in the parameter `clicks` if it wants to recognise single or double clicks and take different action according to which is detected. Higher values could be used to allow the detection of triple clicks, but most users will not be able to click fast enough within the time permitted. The number of times that the state was detected is returned as the function result.

The function also returns the mouse position, the state of all the mouse buttons, and the state of the shift, control and alt keys at the time it returns. The key states may be used to modify the meaning of a mouse click – for example clicking on an object might select it and deselect the previously selected object(s), while clicking with the shift key depressed might select it without deselecting any other selected objects.

4.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
clicks	*WORD*	**Button clicks** This parameter specifies the number of mouse button clicks for which the function should wait; it should normally be set to 1 or 2 depending on whether single clicks only or double and single clicks are required.
mask	*WORD*	**Button mask** This parameter is a bit map specifying which button or buttons should be monitored; thus a value of 0x0001 monitors the left button, 0x0002 the right button, and 0x0003 monitors both buttons. GEM can theoretically support a mouse with up to 16 buttons.
state	*WORD*	**Button state** This parameter is a bit map similar to mask; it is used to indicate for each button specified by mask whether button up (0) or button down (1) is to be detected. For example if mask is set to 0x0003, a state of 0x0001 means that the function should wait until the left button is down and the right button is up.
pmx pmy	*WORD* * *WORD* *	**Mouse coordinates** These parameters point to the objects where the final position of the mouse will be returned, in pixels from the top left corner of screen.
pmb	*WORD* *	**Mouse button state return** This parameter points to a bit map similar to mask which is used to return the state of all the mouse buttons when the function returns. Thus 0x0001 means the left button is down, 0x0002 means the right button is down, and 0x0003 means both buttons are down.

`pks` *WORD ** **Key state return**

This parameter points to an object into which is written the current state of the Shift, Control and Alt keys as a bitmap with the following meanings

0x0001	Right Shift key depressed
0x0002	Left Shift key depressed
0x0004	Control key depressed
0x0008	Alt key depressed

This result is provided so that applications can make use of these keys to modify the result of a mouse action – for example Shift-Click can be used to signify "extend a selected area of text" in a situation where Click is used to reposition the text cursor.

4.2.4 Function Result

The value returned is a WORD specifying the number of times the required state was entered within the double-click delay period.

4.2.5 Example

```
WORD kstate, dummy, mx, my;
WORD nclicks;

/* Wait for the left hand button to be depressed up to
  2 times, ignoring final button state */
nclicks = evnt_button(2, 1, 1,
                      &mx, &my, &dummy, &kstate);

if (nclicks == 2)
  { /* Process a double click */
  }
else if (kstate & 0x0003)
  { /* One or both shift keys down
       - process a shift click */
  }
else
  { /* Process a standard click */
  }
```

4.3 Wait For Mouse Event `evnt_mouse`

Wait For Mouse Event allows an application to watch the position of the mouse as it moves around the screen and take appropriate action. The function watches a specified rectangle; it returns when the mouse enters or leaves that rectangle.

4.3.1 Definition

The Prospero C definition of Wait For Mouse Event is :

```
WORD evnt_mouse(WORD leave, WORD x, WORD y,
                WORD width, height,
                WORD *pmx, WORD *pmy,
                WORD *pmb, WORD *pks);
```

4.3.2 Purpose

This function allows an application to watch the mouse position, and detect when it enters or leaves a specified rectangle. In applications programming mouse watching is used to change the mouse shape to provide visual feedback to the user, and to fulfil GEM's requirement that mouse forms other than the arrow and the busy cursor form are only permitted within the work area of the active window. An example of this can be found in many editors; the user expects a text cursor (a vertical bar) to select text within the document, and a pointer to move around the menu or the control areas of the window. By changing the pointer when it crosses the work area boundary, the application indicates immediately which are control areas and which are writing areas. A simple example of how to use `evnt_mouse` to achieve this is given in section 4.3.5 – note however that for any but the simplest cases or testing purposes, `evnt_multi` will have to be used (section 4.6), as applications normally want to be able to detect other actions (clicks, key presses, menu selections etc.) at the same time as keeping the mouse cursor form correct.

The principles of using the mouse rectangle watch of `evnt_multi` are exactly the same, except that `evnt_multi` allows an application to watch two completely independent rectangles at the same time. This might be used when different areas of a window's work area had different uses, and therefore required different cursor forms. One rectangle would be used to check when the mouse left its current area of the window, while the other would check that the mouse was within the window as a whole.

4.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`leave`	*WORD*	**Enter or leave flag** This parameter specifies whether the function should return when the mouse enters or leaves the watch rectangle specified by the next four parameters. If `leave` has the value zero, `evnt_mouse` returns when the mouse enters the rectangle. If `leave` has the value one it returns when the mouse leaves the rectangle.
`x`	*WORD*	**Watch rectangle X coordinate**
`y`	*WORD*	**Watch rectangle Y coordinate**
`width`	*WORD*	**Watch rectangle width**
`height`	*WORD*	**Watch rectangle height** The coordinates and size of the rectangle to be watched.
`pmx`	*WORD **	**Mouse X coordinate**
`pmy`	*WORD **	**Mouse Y coordinate** These parameters point to objects used to return the final position of the mouse when the function returns.
`pmb`	*WORD **	**Mouse button state return** This parameter points to an object which will contain the current state of the mouse buttons when the function returns. Each bit of the result corresponds to one mouse button, with the least significant bit corresponding to the left hand button. A bit value of 1 indicates the button is down, while 0 means it is up. Thus 0x0001 means the left button is down, 0x0002 means the right button is down, and 0x0003 means both buttons are down

`pks` *WORD ** **Mouse key state return**

This parameter points to an object which is assigned the current state of the Shift, Control and Alt keys as a bitmap with the following meanings :-

0x0001	Right Shift key depressed
0x0002	Left Shift key depressed
0x0004	Control key depressed
0x0008	Alt key depressed

4.3.4 Function Result

The value returned is reserved, and will always equal one.

4.3.5 Example

```
WORD mx, my;                /* Mouse coordinates */
WORD x, y, w, h;            /* work area coordinates */
WORD dummy;                 /* for unwanted extra info */
WORD inside;                /* 0 means outside area */

inside = 0;

/* We could set the initial value according to whether
   the mouse was in the rectangle, but if we just
   guess, it doesn't matter if it is wrong as
   evnt_mouse will return immediately if the return
   condition is already satisfied
*/

  while (1)  /* forever */
    {
      evnt_mouse(inside, x, y, w, h,
                 &mx, &my, &dummy, &dummy);
      inside = !inside;
      /* evnt_mouse returns when the value of inside no
         longer corresponds to the mouse position */
      if (inside)
        graf_mouse(1, 0);          /* Text cursor */
      else
        graf_mouse(0, 0);          /* Arrow */
    }
```

4.4 Wait For Message Event `evnt_mesag`

Wait For Message Event is provided to allow a program to await the arrival of a standard system message. These can be sent by other applications, but are most frequently received from the screen manager to indicate that the user has selected a menu, or operated one of the window control points. See the introduction to section 4 for details of the standard messages.

4.4.1 Definition

The Prospero C definition of Wait For Message Event is :

```
WORD evnt_mesag(WORD pbuff[8]);
```

4.4.2 Purpose

The purpose of Event Message is to wait until the arrival of a standard 16-byte message. The only normal source of messages is the GEM AES screen handler, which notifies the user of events such as menu selection or user interaction with the window control points. This function always returns a 16-byte message – if the message is longer, this is indicated by the second word being non-zero. If an application wants to detect other events as well, such as key presses or mouse clicks, it must use `evnt_multi` (section 4.6). However, many simple applications which simply wait for a menu item to be selected, then perform the required action, will not need to recognise other events, and so will use `evnt_mesag` in the main event loop.

4.4.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`pbuff`	*WORD[8]*	**Message buffer** This parameter provides the location in which the 8-word message received is returned.

4.4.4 Function Result

The value returned is reserved, and will always equal one.

4.4.5 Example

```
#define MN_SELECTED 10  /* In AESBIND.H */
#define tdesk  3        /* Sample constants provided */
#define tfile  5        /* by resource editor */

WORD buffer[8];
WORD quitting = 0;

do {
  evnt_mesag(buffer);
  switch (buffer[0])    /* Contains type of message */
    {
     case MN_SELECTED: switch (buffer[3])
                       /* Contains index of title */
                       {
                        case tdesk: dodesk(buffer[4]);
                                    break;
                        case tfile: dofile(buffer[4]);
                                    break;
                          ...
                       /* One function per title */
                       }
     /* could also handle window messages */
    }
   } while (quitting == 0);
   /* quitting would be set to 1 by dofile if QUIT
      was selected */
```

4.5 Wait For Timer Event `evnt_timer`

Wait For Timer Event is used to wait for a specified length of time. It should only be used when a pure delay function is required – for an interruptible timer, or to place a time limit on other functions the timing facilities in `evnt_multi` should be used (see section 4.6).

4.5.1 Definition

The Prospero C definition of Wait For Timer Event is :

```
WORD evnt_timer(WORD locnt, WORD hicnt);
```

4.5.2 Purpose

This function allows an application to wait for a specified length of time. This in itself is unlikely to be useful very often, though when combined with other event waiting in `evnt_multi` (section 4.6) its usefulness greatly increases. Note that any other application which is ready to run (such as a desk accessory) may be brought into context when this call is made, which means that the delay may be longer than that requested. This function could also be used simply in order to allow other applications to come into context, by specifying a very short delay. GEM version 2.0 provides a better way of achieving this using `appl_yield` (section 3.7).

4.5.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`locnt`	*WORD*	**Millisecond timer (low word)**
`hicnt`	*WORD*	**Millisecond timer (high word)**
		These parameters are used to specify the period in milliseconds for which the function should wait.

4.5.4 Function Result

The value returned is reserved, and will always equal one.

4.5.5 Example

```
evnt_timer(1000, 0); /* Wait 1 second */
.
.
evnt_timer(1, 0);    /* Allow other processes a go */
```

4.6 Wait For Multiple Events `evnt_multi`

Wait For Multiple Events is probably the most important function in GEM AES, and forms the core of every serious application. This is a do-everything function – the Swiss Army Knife of GEM event handling, and as a result it has an excessive number of parameters and returns a lot of results. If only one type of event is required, an application should by preference use one of the following functions instead:

Wait for a keystroke	`evnt_keybd`	section 4.1
Wait for a mouse click	`evnt_button`	section 4.2
Watch for mouse movements	`evnt_mouse`	section 4.3
Wait for a standard message	`evnt_mesag`	section 4.4
Wait a specified time	`evnt_timer`	section 4.5

Only if an application wants to detect more than one of these events should it use `evnt_multi`.

4.6.1 Definition

The Prospero C definition of Event Multi is :

```
WORD evnt_multi(WORD flags,
                WORD bclk, WORD bmsk, WORD bst,
                WORD m1leave,
                WORD m1x, WORD m1y, WORD m1w, WORD m1h,
                WORD m2leave,
                WORD m2x, WORD m2y, WORD m2w, WORD m2h,
                WORD mepbuff[8],
                WORD tlc, WORD thc,
                WORD *pmx, WORD *pmy, WORD *pmb,
                WORD *pks, WORD *pkr, WORD *pbr);
```

4.6.2 Purpose

The purpose of Event Multi is to wait until an event happens. This can be one or more of up to six different sorts of event, each specified by a particular bit in the parameter `flags` as follows:

Event	Value	Name
a key being pressed	0x0001	`MU_KEYBD`
mouse button change	0x0002	`MU_BUTTON`
mouse entering or leaving a rectangle	0x0004	`MU_M1`
mouse entering or leaving another rectangle	0x0008	`MU_M2`
the arrival of a standard system message	0x0010	`MU_MESAG`
the expiration of a time delay	0x0020	`MU_TIMER`

The above values are provided as named constants in the file AESBIND.H; they can be combined using the | operator to indicate what combination of events the function should wait for.

A normal GEM based application will initialize itself, perhaps open a window or two, then perform an `evnt_multi` call to wait for further instructions from the user. Whenever an event is received, the application decides what the event was and what the user wanted done, and goes away and does it. When this is complete, the application returns to the main loop containing the `evnt_multi` call to see what the user wants to do next. Thus at least one `evnt_multi` call is likely to be at the heart of any GEM based program which makes proper user of the desktop; if you don't have a window and don't allow Desk Accessories, programming will certainly be easier, but it will hardly be GEM.

The purpose and typical usage of each of the events is described further in the relevant sub-section. However, the ability to wait for a time to elapse takes on a new usefulness when combined with other event waiting. For example, an application can use this to put a time limit on other events, so that if nothing happens within a certain time the user gets a reminder that they are expected to make a response. Alternatively, an application doing some long calculation could check periodically to see if the user has clicked on a button labelled stop, using the timer event to return quickly and continue calculating if no click has occurred.

4.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`flags`	*WORD*	**Event flags** This parameter is used to indicate what events the application is waiting for. Each bit corresponds to one of the possible events, as described under 4.6.2 above. Often the value passed will be a combination (using `|`) of the relevant constants from the AESBIND.H file – if a particular combination is used frequently it can be declared as a new constant with the relevant bits set. It is also sometimes useful to pass a variable, so that the events enabled depend upon the state of the program. For example, an application might only be interested in button events when the mouse is within the window work area.
`bclk`	*WORD*	**Button clicks** This parameter is used when button events are enabled – it specifies the number of times a button state is to be entered before returning. See the description of the parameter `clicks` in section 4.2.3 for further information.
`bmsk`	*WORD*	**Button mask** This parameter is used when button events are enabled – it specifies which mouse buttons are to be checked. See the description of the parameter `mask` in section 4.2.3 for further information.
`bst`	*WORD*	**Button state** This parameter is used when button events are enabled – it specifies the state (up or down) for which each mouse button is to be checked. See the description of the parameter `state` in section 4.2.3 for further information.

`m1leave` *WORD* **First watch rectangle leave flag**

This parameter is used when mouse events are enabled for the first mouse rectangle – it specifies whether the application is waiting for the mouse to leave or enter the rectangle. See the description of the parameter `leave` in section 4.3.3 for further information.

`m1x` *WORD* **First watch rectangle X coordinate**
`m1y` *WORD* **First watch rectangle Y coordinate**
`m1w` *WORD* **First watch rectangle width**
`m1h` *WORD* **First watch rectangle height**

These parameters are used when mouse events are enabled for the first mouse rectangle – they specify the coordinates and size of the rectangle being watched. See the descriptions of the parameters `x`, `y`, `width` and `height` in section 4.3.3 for further details.

`m2leave` *WORD* **Second rectangle leave flag**

This parameter is used when mouse events are enabled for the second mouse rectangle – it specifies whether the application is waiting for the mouse to leave or enter the rectangle. See the description of the parameter `leave` in section 4.3.3 for further information.

`m2x` *WORD* **Second rectangle X coordinate**
`m2y` *WORD* **Second rectangle Y coordinate**
`m2w` *WORD* **Second rectangle width**
`m2h` *WORD* **Second rectangle height**

These parameters are used when mouse events are enabled for the second mouse rectangle – they specify the coordinates and size of the rectangle being watched. See the descriptions of the parameters `x`, `y`, `width` and `height` in section 4.3.3 for further details.

`mepbuff` *WORD [8]* **Message buffer**

This parameter is used when message events are enabled – it provides the location in which messages received are returned.

`tlc` *WORD* **Millisecond timer (low word)**
`thc` *WORD* **Millisecond timer (high word)**

These parameters are used when timer events are enabled – they specifies the number of milliseconds for which `evnt_multi` should wait, if no other enabled event occurs first.

`pmx` *WORD ** **Mouse X coordinate**
`pmy` *WORD ** **Mouse Y coordinate**

These parameters point to the objects used to receive the final position of the mouse on return from `evnt_multi`, whatever the event was that caused it to return. They thus combine the functions of the parameters `pmx` and `pmy` of both `evnt_button` (section 4.2) and `evnt_mouse` (section 4.3).

`pmb` *WORD ** **Mouse button state return**

This parameter points to the object used to receive the final state of the mouse buttons on return from `evnt_multi`, whatever the event was that caused it to return. It thus combines the functions of the parameter `pmb` of both `evnt_button` (section 4.2) and `evnt_mouse` (section 4.3).

`pks` *WORD ** **Keyboard state return**

This parameter points to the object used to receive the final state of the control, shift and ALT keys on return from `evnt_multi`, whatever the event was that caused it to return. It thus combines the functions of the parameter `pks` of both `evnt_button` (section 4.2) and `evnt_mouse` (section 4.3).

`pkr` *WORD ** **Key pressed**

This parameter points to the object used to return the value of the key pressed, when keyboard events are enabled. An object must be provided to receive this value, even if keyboard events are not enabled. The value stored is only valid if keyboard events are enabled, and a keyboard event in fact happened – this can be determined by testing the MU_KEYBD bit of the `evnt_multi` function result. The value stored in the object gives both the ASCII code and scan code of the key pressed, in the low and high bytes respectively, equivalent to the value returned by `evnt_keybd` – see section 4.1.4 for more details.

`pbr` *WORD ** **Button clicks**

This parameter points to the object used when button events are enabled to return the number of times the specified state was entered within the double click delay period. An object must be provided to receive this value, even if button events are not enabled. The value is only valid if button events are enabled, and a button event did in fact happen – this can be determined by testing the MU_BUTTON bit of the `evnt_multi` function result. The value stored is equivalent to that returned by `evnt_button` – see section 4.2.4 for more details.

4.6.4 Function Result

The function result is a bitmap indicating which event or events caused the function to return, using the same bit values as for the parameter `flags`. The application can test each bit by seeing whether, for example `((result & MU_KEYBD) != 0)`.

4.6.5 Example

It is difficult to give a short example of the use of `evnt_multi`; a complete program using `evnt_multi` is included with the software issue disk.

However, the following example shows a simple example of using two `evnt_multi` features together. It uses `evnt_multi` to wait for a keystroke for one second only :-

```
#define MU_KEYBD 0x0001  /* From AESBIND.H file */
#define MU_TIMER 0x0020

WORD which, key, dummy;
WORD dummy_buffer[8];
.
.
/* Wait 1 second for a key to be pressed */
which = evnt_multi(MU_KEYBD | MU_TIMER,
        0,0,0,          /* MU_BUTTON parameters */
        0,0,0,0,0,      /* MU_M1 parameters */
        0,0,0,0,0,      /* MU_M2 parameters */
        dummy_buffer,   /* for MU_MESAG*/
        1000, 0         /* return after 1 sec
                           if no key */
        &dummy,&dummy,  /* mouse coords not used */
        &dummy,&dummy,  /* state parameters not used */
        &key,           /* the key that was pressed */
        &dummy);        /* clicks return not used */

if (which & MU_KEYBD)
  { /* deal with the keystroke */
  }
```

4.7 Set Double Click Delay `evnt_dclick`

Set Double Click Delay is used to set or discover the length of time during which GEM waits after a mouse button click to count extra clicks and so allow applications to determine if there has been a double click. See also the description of `evnt_button` in section 4.2.

4.7.1 Definition

The Prospero C definition of Set Double Click Delay is :

```
WORD evnt_dclick(WORD rate, WORD setit);
```

4.7.2 Purpose

This function may be used to select one of five possible time delays for detecting double clicks, or to return the current setting. Experienced users tend to prefer faster double click speeds (i.e. shorter delays), as they are used to using the mouse, and tend to make single clicks more quickly, which might be mistaken for double clicks if a longer delay was set. Less experienced users find it hard to click fast enough if the double click delay is too short. This function might be used to allow the user to select a double click delay, perhaps in a control panel type desk accessory, or a particularly user-friendly application.

4.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`rate`	*WORD*	**New double click delay** A value in the range 0 to 4, where 0 corresponds to a long delay (slow double clicks recognised) and 4 to a short delay (only fast double clicks recognised).
`setit`	*WORD*	**Set or inquire flag** If the value of this parameter is one, the new double click speed will be set to the value in the parameter `rate`. Otherwise, the value of `rate` is ignored. This would be used if the application wanted to discover the double click speed without altering it.

4.7.4 Function Result

The value returned will be the current setting of the double click speed, using the same values as for the parameter `rate`. If `setit` was one, the value returned will be the newly set rate, otherwise it will be the old (and still current) rate.

4.7.5 Example

```
if (evnt_dclick(1, 0) == 4) /* Get current rate */
  evnt_dclick(3, 1);        /* Allow any rate but 4 */
```

5 MENU LIBRARY

This section contains descriptions of the Menu Library functions, in the following sub-sections.

Section	Function description	Binding name
5.1	Display Menu Bar	`menu_bar`
5.2	Check Menu Item	`menu_icheck`
5.3	Enable Menu Item	`menu_ienable`
5.4	Menu Title Display	`menu_tnormal`
5.5	Alter Menu Text	`menu_text`
5.6	Register Accessory	`menu_register`
5.7	Unregister Accessory	`menu_unregister`
5.8	Create Menu Bar	`menu_create`
5.9	Add Menu Title	`menu_title`
5.10	Add Menu Item	`menu_item`

The routines in the Menu Library are concerned with controlling the appearance of the application's menu bar.

The application is not responsible for interaction between the mouse and the menu bar – this is performed by the GEM AES screen manager – but it is up to the application to control which (if any) menu is to be displayed (`menu_bar`), and to alter the menu according to context, by for example placing or removing a tick mark by an item to indicate whether the corresponding option is selected (`menu_icheck`), disabling certain selections when their selection is not appropriate (`menu_ienable`), or altering the text of a menu item to reflect the current conditions (`menu_text`). The application is also responsible for restoring the title of a selected menu item to normal video when the processing of that menu selection is complete (`menu_tnormal`).

The menu bar is stored in the form of an object tree, as described in section 6. By convention, this is the first object tree in an application's resource file, and the pointer to it can therefore be obtained by giving `rsid` a value of zero when calling `rsrc_gaddr` (section 12.3). If an application does not want to use a resource file, a menu tree can be created dynamically using the `menu_create`, `menu_title` and `menu_item` functions described in sections 5.8 onwards. These functions do not form part of the original bindings provided by Digital Research, and do not make any calls to the GEM AES, but are provided by Prospero Software to assist in the creation of menu bars, which is otherwise somewhat complicated.

The functions `menu_register` and `menu_unregister` are for use by desk accessories, to control the name or names which appear in the accessory menu for that accessory application.

5.1 Display Menu Bar `menu_bar`

Display Menu Bar makes GEM AES enable and draw the menu bar at the top of the screen, or disable the menu bar.

5.1.1 Definition

The Prospero C definition of Display Menu Bar is :

```
WORD menu_bar(OBJECT *tree, WORD showit);
```

5.1.2 Purpose

This function is used to tell GEM AES whether the menu bar is to be displayed or not, and whether the mouse is to interact with the menu bar to produce drop down menus. The pointer to the tree data structure containing the menu bar is passed in the parameter `tree` – this will normally have been obtained from `rsrc_gaddr` (section 12.3) after loading the resource file, or using `menu_create` (section 5.8) if a resource file is not used. Refer to section 6 for a full description of object trees and the `OBJECT` type. It is possible to select which menu to use (if there is more than one available) by using this function to make a particular menu bar active. If the application requests that the menu bar be made active, by making `showit` non-zero, GEM AES will draw the menu bar at the top of the screen. However if `showit` is zero, the menu bar will not be explicitly erased, but simply made inactive, and not redrawn should it become obscured. Normally the menu will only be removed when an application is about to terminate.

5.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT **	**Object tree containing menu** The tree pointer of the object tree containing the menu to be enabled (and displayed) or disabled. This will normally be obtained using `rsrc_gaddr` (section 12.3) or `menu_create` (section 5.8).

`showit` *WORD* **Enable or disable flag**

If this parameter is one, the specified menu bar will be made active, and drawn at the top of the screen. If it is zero, the menu will be disabled, so that the mouse no longer interacts with the menu when an `evnt_mesag` (section 4.4) or `evnt_multi` (section 4.6) call is made, and the menu bar is not redrawn when the screen is refreshed.

5.1.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.1.5 Example

```
#define TheMenu  1    /* Sample index in resource file,
                         returned by resource editor */

OBJECT *MyMenu;

main()
{
  /* Initialize application, load resource file */
  ...
  /* Get pointer to menu bar */
  rsrc_gaddr(0, TheMenu, &MyMenu);

  /* Display and enable menu bar */
  menu_bar(MyMenu, 1);
  .
  /* Disable menu before terminating */
  menu_bar(MyMenu, 0);
  rsrc_free();
  appl_exit();
}
```

5.2 Check Menu Item `menu_icheck`

Check Menu Item is used to control whether a menu item is displayed with or without a check mark at the left hand end – this can be used by an application to indicate whether an option is currently in force, for example. In GEM version 1.1, the check mark is a tick, while in GEM version 2.0 an arrowhead is used.

5.2.1 Definition

The Prospero C definition of Check Menu Item is :

```
WORD menu_icheck(OBJECT *tree,
                 WORD itemnum, WORD checkit);
```

5.2.2 Purpose

This function is used to tell GEM AES whether the specified menu item is to be displayed checked or unchecked (with or without a check mark). If the menu item is displayed at the time, it will be redrawn in the new state. The pointer to the tree containing the menu bar is passed in the parameter `tree` – usually obtained from `rsrc_gaddr` (section 12.3) after loading the resource file, or `menu_create` (section 5.8) if a resource file is not being used. The object index within the tree of the item to be checked or unchecked is passed in the parameter `itemnum`; this is provided as a macro in the include file produced by the resource editor, or returned by `menu_item` (section 5.10) if a resource file is not being used.

5.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Object tree containing menu** The tree pointer of the object tree containing the menu concerned. This will normally be obtained using `rsrc_gaddr` (section 12.3) or `menu_create` (section 5.8).

`itemnum` *WORD* **Menu item index**

This parameter gives the index within the menu tree array (see section 6) of the item to be checked or unchecked. It will normally be a macro provided by the resource editor in an include file, or the result returned by a call to `menu_item` (section 5.10).

`checkit` *WORD* **Item check flag**

If this parameter is one, the specified menu item will be displayed with a check mark beside it. If zero, the item is displayed without a check mark. It is not an error to request a check mark on an item which is already checked.

5.2.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.2.5 Example

```
#define ifast 13     /* Sample index in menu tree of
                        menu item ' Fast Mode' from
                        resource editor */

OBJECT *MyMenu;
WORD FastMode;       /* My record of mode */

  .
  .
  /* Make menu indicate what mode we are in */
  menu_icheck(MyMenu, ifast, FastMode);
```

5.3 Enable Menu Item `menu_ienable`

Enable Menu Item makes GEM AES enable or disable a specified item or title in the menu bar. Disabled items are displayed in light text, and cannot be selected using the mouse.

5.3.1 Definition

The Prospero C definition of Enable Menu Item is :

```
WORD menu_ienable(OBJECT *tree,
                  WORD itemnum, WORD enableit);
```

5.3.2 Purpose

This function is used to tell GEM AES whether the specified menu item is enabled or disabled. Normally an application will disable any items which are not relevant at the current state of the application; such items are displayed in light text, and do not interact with the mouse when it passes over them. In GEM version 2.0, menu titles can be enabled or disabled as well as menu items. The effect of a call to `menu_ienable` specifying the index of a title rather than an item is not documented in GEM version 1.1.

5.3.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT* *	**Object tree containing menu** The tree pointer of the object tree containing the menu concerned. This will normally be obtained using `rsrc_gaddr` (section 12.3) or `menu_create` (section 5.8).

`itemnum` *WORD* **Menu item index**

The index within the menu tree of the item to be enabled or disabled. This will normally be given by a macro from the include file created by the resource editor when the resource file was created. In GEM version 2.0, menu titles may be specified – in this case the high order bit of the title index specified should be set. The simplest way to do this is by passing a value of `(title_index | 0x8000)`.

`enableit` *WORD* **Enable or disable flag**

If this parameter is one, the specified menu item will be enabled, and drawn in normal type. If it is zero, the menu item will be displayed in gray, and disabled, so that the item or title can not be selected using the mouse.

5.3.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.3.5 Example

```
#define saveitem 5    /* Sample index of menu item
                         ' Save' in header file from
                         resource editor */

OBJECT *my_menu;
WORD alteredflag;

  /* Only allow user to save if work altered */
  menu_ienable(my_menu, saveitem, alteredflag);
```

5.4 Menu Title Display `menu_tnormal`

Menu Title Display makes GEM AES display a menu title in either normal or reverse video. This is conventionally used in GEM applications to indicate when a selection from a particular menu title's list is being processed.

5.4.1 Definition

The Prospero C definition of Menu Title Display is :

```
WORD menu_tnormal(OBJECT *tree,
                  WORD titlenum, WORD normalit);
```

5.4.2 Purpose

This function is used to tell GEM AES whether the specified menu title is to be displayed in normal or reverse video. This gives an indication to the user about what the application is doing. By convention, when an application is waiting for menu input, all menu items will be displayed in normal video. When a menu item is selected with the mouse, GEM AES sends a message to the application indicating which menu item and title were selected (see `evnt_mesag` and `evnt_multi` in section 4). The title of the selected menu item will be left in reverse video state by GEM AES, and it is up to the application to reset it to normal at some stage before waiting for another menu selection. The normal approach is to reset the title immediately prior to the next `evnt_mesag` or `evnt_multi` call, so that it remains highlighted all the time the menu selection is being processed, and the user can tell that the application is busy.

If an application makes some menu choices available by typing keys at the keyboard as well as by selecting them with a mouse, it is a good idea to highlight the relevant menu title when the key press is detected, to indicate that the selection has been made. This will not be done automatically be GEM AES, so the application should call `menu_tnormal` with a value of zero in the parameter `normalit` when the keystroke is detected.

5.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Object tree containing menu** The tree pointer of the object tree containing the menu concerned. This will normally be obtained using `rsrc_gaddr` (section 12.3) or `menu_create` (section 5.8).
`titlenum`	*WORD*	**Menu title index** The index within the menu tree of the title to be displayed highlighted or normal. The title of the menu item selected is given in `pbuff[3]` when a menu selection message is returned by `evnt_mesag` or `evnt_multi` – this title should always be returned to normal text before awaiting further message events.
`normalit`	*WORD*	**Normal or highlighted flag** If this parameter is one, the specified menu title will be displayed in normal text. If zero, the title will be highlighted by drawing it in reverse video.

5.4.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.4.5 Example

```
#define MN_SELECTED 10     /* From AESBIND.H */

#define newitem  3
#define saveitem 4
#define quititem 5
    /* Sample constants from resource editor */

OBJECT *MyMenu;
WORD MyBuffer[8];

do {
    /* Wait for message events */
    evnt_mesag(MyBuffer);
    if (MyBuffer[0] == MN_SELECTED)
      {                          /* Menu item selected */
        switch (MyBuffer[4])   /* Which item selected */
        { case saveitem: ...
          case newitem:  ...
        } /*switch*/

        /* Now reset title to normal */
        menu_tnormal(MyMenu, MyBuffer[3], 1)
      } /*IF MN_SELECTED */
   } while (MyBuffer[4] != quititem);
```

5.5 Alter Menu Text `menu_text`

Alter Menu Text is used to alter the text of a menu item. This is frequently used by applications to give context-sensitive menus, so that a menu item might read ' Enable option' when an option was disabled and ' Disable option' when it was enabled. In this way the text of the menu item also indicates the current state of the option.

5.5.1 Definition

The Prospero C definition of Alter Menu Text is :

```
WORD menu_text(OBJECT *tree
               WORD inum, const char *ptext);
```

5.5.2 Purpose

This function is used to alter the text of the specified menu item. The application should ensure that the new text is no longer than the original text or GEM is liable to crash. Menu items of more than about 20 characters are unlikely to be useful, especially as a menu is not permitted to occupy more than a quarter of the screen area.

This function does not cause the menu to be redrawn, so if there is any possibility of the specified menu item being currently displayed (this is only possible if using `evnt_multi`, with message events and some other event type enabled, where the other event has caused a return from `evnt_multi` while a menu was being held down), the application can call `menu_bar` (section 5.1) to redraw the menu.

5.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT **	**Object tree containing menu** The tree pointer of the object tree containing the menu concerned. This will normally be obtained using `rsrc_gaddr` (section 12.3) or `menu_create` (section 5.8).

inum	*WORD*	**Menu item index** The index within the menu tree of the menu item whose text is to be altered. This will normally be a macro from the include file generated by the resource editor when the resource file was created.
ptext	*const char* *	**New menu text** The new text of the menu item. This must be no longer than the original text when the menu was created, and terminated by a null character.

5.5.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.5.5 Example

```
#define optitem  20
    /* Sample constant from resource editor */

OBJECT *MyMenu;
WORD OptionOn;

  if (OptionOn)
    menu_text(MyMenu, optitem, "  Disable option");
  else
    menu_text(MyMenu, optitem, "  Enable option ");
```

5.6 Register Accessory menu_register

Register Accessory is used by a desk accessory to add a name to the list of accessories on the desk menu.

5.6.1 Definition

The Prospero C definition of Register Accessory is :

```
WORD menu_register(WORD pid, const char *pstr);
```

5.6.2 Purpose

This function is used to add the name of a desk accessory to the list of available accessories in the Desk menu at the end of the menu bar. In GEM version 1.1, this menu conventionally has the title Desk, and is at the left hand corner of the screen. In GEM version 2.0 the title is altered by GEM AES to the name of the currently running application, and placed on the right hand end of the menu bar, but this does not affect the application.

The desk accessory must give its application global identifier (returned by `appl_init` – see section 3.2) in the parameter `pid` – this indicates where messages will be sent when the accessory's menu item is selected. The text to be added to the desk menu must also be given. Note that one desk accessory file may contain more than one desk accessory – in this case it will make several calls to `menu_register` to add the name of each accessory before waiting for a message. The function result returns the menu item identifier of the accessory slot allocated, which will enable the accessory file to discover which menu item was selected when it receives a message, and therefore decide what function to perform. If a desk accessory program only adds one item to the desk menu, the menu item identifier will not be necessary as all menu selection messages must refer to that item. However, it is prudent to check that the value returned was not –1, which is used to indicate that the desk menu is full – if this occurs there is no point in an accessory waiting for a message, and it may as well terminate.

5.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`pid`	*WORD*	**Accessory global identifier** The application global identifier of the desk accessory process, as returned by `appl_init` at the start of the desk accessory program.
`pstr`	*const char **	**Desk menu entry** The text which the accessory wishes to place in the desk menu describing the accessory function it offers. GEM AES remembers the address of this text rather than its contents, so it must not be modified after being passed to this function. (a string literal is particularly suitable).

5.6.4 Function Result

The value returned will be –1 if the desk menu is full, otherwise a value in the range 0 to 5 (this value is likely to increase in subsequent issues of GEM) indicating which menu slot was allocated to the identifier. This is the value returned in `pbuff[3]` when an AC_OPEN message is received by `evnt_mesag` (see section 4.4) or `evnt_multi` (section 4.6) indicating an accessory has been opened, and may be used to tell which accessory has been requested if more than one item has been added to the desk menu.

5.6.5 Example

```
WORD ap_id, calc_id, clock_id;
WORD my_buffer[8];

calc_id  = menu_register(ap_id, "  Calculator");
clock_id = menu_register(ap_id, "  Clock");
if (calc_id == -1) exit(3);   /* No room for either */

do {
     evnt_mesag(my_buffer);
     switch (my_buffer[0])
     { case AC_OPEN: if (my_buffer[3] == calc_id)
                       { /* Do calculator */ }
                     else
                       { /* Do clock*/ }
                     break;
       case AC_CLOSE: ...
     }
   } while (1);          /* No need to terminate */
```

5.7 Unregister Accessory `menu_unregister`

Unregister Accessory is used by a desk accessory to remove a name from the list of accessories on the desk menu. This function is not available in GEM version 1.1.

5.7.1 Definition

The Prospero C definition of Unregister Accessory is :

```
WORD menu_unregister(WORD mid);
```

5.7.2 Purpose

This function is used to remove the name of a desk accessory from the list of available accessories in the Desk menu at the end of the menu bar. It should only be used by desk accessories, and is not available in GEM version 1.1. The menu item identifier of the name to be removed must be specified; this is the value returned by `menu_register` when the name was added to the desk menu. A value of –1 can be used to indicate the desk accessory name belonging to the currently running process. The effect of this when a process has added more than one name is not documented.

5.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`mid`	*WORD*	**Menu item identifier** The menu item identifier of the name to be removed, as returned by `menu_register`. A value of –1 may be used to indicate the name belonging to the current process, when only a single name has been added.

5.7.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

5.7.5 Example

```
WORD ap_id, calc_id, clock_id;

calc_id  = menu_register(ap_id, "  Calculator");
clock_id = menu_register(ap_id, "  Clock");
   .
   .
menu_unregister(clock_id); /* Remove clock from menu */
```

5.8 Create Menu Bar `menu_create`

Create Menu Bar is used to create a menu bar at run time rather than loading one from a resource file. The use of a resource file is recommended for several reasons – it will result in smaller applications, and ones which are easily translated to foreign languages. However, where these considerations are outweighed by the desire to have the application self contained in a single file, or for simple applications, the use of this function may be preferable.

This function does not form part of the original bindings provided by Digital Research.

5.8.1 Definition

The Prospero C definition of Create Menu Bar is :

```
OBJECT *menu_create( WORD titles, WORD items,
                     const char *about);
```

5.8.2 Purpose

This function is used to allocate space for a menu object tree, and set up the objects within the tree so that they constitute a valid menu bar. The Desk menu title, whose index is always 3, and must be present on all menu bars, is created by this function, and the text for the single user-defined item on this menu must be supplied in the parameter `about`. This usually takes the form ' About programname ...', and when selected should result in a dialog giving details of the program, such as the version number and copyright message. The remainder of this menu is reserved for desk accessories.

The menu bar can be used immediately after creating it, though there would only be one title with one item to be selected (apart from the desk accessories). Alternatively, further titles and menu items can be added to the tree using `menu_title` (section 5.9) and `menu_item` (section 5.10). The application must specify how many titles and items are to be added to the tree in this way, so that sufficient space is allocated for the tree. The value in the parameter `titles` need not include the Desk title, and the value in the parameter `items` need not include the items associated with the Desk title, but the values should take account of all titles and items added using `menu_title` and `menu_item`.

5.8.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`titles`	*WORD*	**Number of titles to be added** The number of titles to be added to the menu. The application should add 1 for each title added using `menu_title` (section 5.9). There is no need to allow for the Desk title, but it is a good idea during program development to add a bit extra for menus which will be added later.
`items`	*WORD*	**Number of items to be added** The number of items to be added to the tree. The application should add 1 for each item added using `menu_item` (section 5.10). There is no need to allow for the Desk items, but it is a good idea during program development to add a bit extra for menus which will be added later.
`about`	*const char **	**About menu item text** The text of the 'About...' menu item – the first item on the Desk menu above the desk accessories.

5.8.4 Function Result

The function returns a pointer to the newly created tree, which can then be passed to `menu_bar` or any of the other routines in this section (though normally an application will add some additional titles and items before using the menu).

5.8.5 Example

```
OBJECT  *MyMenu;
WORD File_title, Play_title;
WORD Quit_item, Go_item, Stop_item;

  MyMenu = menu_create(2, 3, "  About Myprog");
  File_title = menu_title(MyMenu, " File ");
  Play_title = menu_title(MyMenu, " Play ");
  Quit_item = menu_item(MyMenu, File_title, " Quit ");
  Go_item   = menu_item(MyMenu, Play_title, " Go ");
  Stop_item = menu_item(MyMenu, Play_title, " Stop ");
  /* The menu is now ready for use */
```

5.9 Add Menu Title `menu_title`

Add Menu Title is used to add a title to a menu created using `menu_create` (section 5.8). The Desk menu is added automatically by `menu_create`, as it is required on all menus; other menu titles must be added by the application. Titles should be added in the order in which they are to appear from left to right.

This function does not form part of the original bindings provided by Digital Research.

5.9.1 Definition

The Prospero C definition of Add Menu Title is :

```
WORD menu_title(OBJECT *menu, const char *title);
```

5.9.2 Purpose

This function is used to add a title to a menu bar created using `menu_create` (section 5.8). All titles should be added before any items are added using `menu_item` (section 5.10). It is a good idea to put a space at each end of the title string – this will separate the titles on the menu bar, and cause the drop-down box for the title to appear slightly indented from the title.

5.9.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
menu	*OBJECT* *	**Object tree containing menu** The tree pointer of the object tree containing the menu, as returned by `menu_create` (section 5.8).
title	*const char* *	**New menu title** A pointer to a null terminated string giving the text of the new menu title. This should begin and end with a space. It is a good idea to keep the total length of all titles below 32 characters where possible, otherwise the menu will not fit on a 40 character display.

5.9.4 Function Result

The function returns the index of the title which was added. This is the value which will be returned in `pbuff[3]` when a message of type MN_SELECTED is received via `evnt_mesag` (section 4.4) or `evnt_multi` (section 4.6), indicating that an item from that menu has been selected.

5.9.5 Example

See section 5.8.5.

5.10 Add Menu Item `menu_item`

Add Menu Item is used to add an item to a menu created using `menu_create` (section 5.8). The ' About ...' item on the Desk menu is added automatically by `menu_create`, as it is required on all menus; other menu items must be added by the application. All menu titles must be added using `menu_title` (section 5.9) before any items are added, and all items must be added to each title, in the order that the titles were added, before adding items to the next title.

This function does not form part of the original bindings provided by Digital Research.

5.10.1 Definition

The Prospero C definition of Add Menu Item is :

```
WORD menu_item(OBJECT *menu, WORD title,
               const char *item);
```

5.10.2 Purpose

This function is used to add a menu item to the specified title in a menu created using `menu_create` (section 5.8). The items are added after all titles are in place, and must be added in the same order as the titles. Failure to observe this rule will result in the values returned by previous calls of `menu_item` no longer being valid. Menus are filled from the top downwards.

The width of each drop-down menu is automatically determined by the longest item added – this item should therefore have a space at the end (unless it is a dividing line), but shorter items need not. Menu items other than dividing lines should begin with one or two spaces. Dividing lines used to separate groups of items in a single menu are made by adding an item whose text is a row of hyphens of suitable length, then disabling the item by a call of `menu_ienable` (section 5.3). If the text of an item is to be altered later using `menu_text` (section 5.5), the length of the text in the parameter `item` should be sufficient to cover any text which will be used, to ensure that enough memory is allocated. The text can be given trailing spaces if required.

Note that in theory up to 24 items can be added to a single menu title, but care must be taken that the area occupied by a menu does not exceed a quarter of the display, or GEM will behave unpredictably, and probably crash.

5.10.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`menu`	*OBJECT **	**Object tree containing menu** The tree pointer of the object tree containing the menu, as returned by `menu_create` (section 5.8).
`title`	*WORD*	**Menu title index** The index within the menu tree of the title to which this item is to be added, as returned by `menu_title` (section 5.9).
`item`	*const char **	**Menu item text** A pointer to a null terminated string giving the text of the new menu item.

5.10.4 Function Result

The function returns the index of the item which was added. This is the value which will be returned in `pbuff[4]` when a message of type MN_SELECTED is received via `evnt_mesag` (section 4.4) or `evnt_multi` (section 4.6), indicating that the item has been selected. The menu index can also be passed to the functions `menu_ienable`, `menu_text`, `menu_icheck` etc.

5.10.5 Example

See section 5.8.5.

6 OBJECT LIBRARY

This section contains descriptions of the Object Library functions, in the following sub-sections.

Section	Function description	Binding name
6.1	Add Object to Tree	`objc_add`
6.2	Delete Object from Tree	`objc_delete`
6.3	Draw Objects in Tree	`objc_draw`
6.4	Find Object under Point	`objc_find`
6.5	Calculate Object Offset	`objc_offset`
6.6	Alter Object Order	`objc_order`
6.7	Edit Text Object	`objc_edit`
6.8	Change Object State	`objc_change`
6.9	Return Object State	`objc_state`
6.10	Set Object State	`objc_newstate`
6.11	Return Object Flags	`objc_flags`
6.12	Set Object Flags	`objc_newflags`
6.13	Return Object Text	`objc_text`
6.14	Set Object Text	`objc_newtext`
6.15	Read or Write Object Header	`objc_read` `objc_write`
6.16	Create Object Tree	`objc_create`
6.17	Insert Item into Object Tree	`objc_item`
6.18	Initialize Editable Text Object	`objc_tedinfo`

Note that the functions from `objc_state` onwards do not form part of the original Digital Research bindings, but are provided in the Prospero C bindings to manipulate the object library data structures which are otherwise slightly awkward.

The word "object" is a term used in GEM AES to mean any of a number of different kinds of graphic images that can appear on the screen. These are described in detail later on, but include various types of boxes, text strings, icons, and combinations of the above. The purpose of the object library is to divide any screen that the programmer wants to see into objects that GEM AES can draw and react to. The objects are organized into a data structure known as an object tree, which allows various objects in the tree to relate to other objects in a specific manner. Although the idea of object trees may appear quite complex at first, it does provide both the programmer and GEM AES with an organized and effective way of dealing with a screen that may contain a hundred different 'things'.

The Object Tree

It is important to understand the object tree data structure before full use can be made of the facilities in the object and form libraries. This data structure is used in GEM AES to describe dialog boxes, menus, forms, and even the desktop background, and may be used by an application to describe complex structures of boxes, text and so on that the application might want to place in a window.

In order to economize on the memory required for pointers, object trees are stored in arrays, so that each link connecting an object to the next object in the tree need only be an array index (taking one word of storage) rather than a pointer to a memory address (requiring two words). The tree is referred to by means of the address of the start of the array – this is held in a variable of type *OBJECT* *. Normally the address will be returned by `rsrc_gaddr` (section 12.2) or `objc_create` (section 6.16). Each element of the array is of type *OBJECT* – there may be up to 32767 elements in a single tree, though as each requires 24 bytes of storage, such a large tree would not fit on many machines. A more normal size of tree would be anything from 10 to 50 objects – for example a tree which contains a menu has 2 objects per title, plus one object per menu item, plus about 5 objects overhead. The array index of a particular element in the array containing a tree is known as the object index, and is of type *WORD*. The header of an object whose object index is `obj` can thus be referenced as `tree[obj]`.

The relevant type definitions from the AESBIND.H file are as follows:

```
typedef struct
 { WORD ob_next, ob_head, ob_tail,
        ob_type, ob_flags, ob_state;
   union
    { void * ptr;       /* Can point to various types */
      unsigned long color;   /* or give color info */
    } ob_spec;
   WORD ob_x, ob_y, ob_width, ob_height;
 } OBJECT;
```

The object header describes the type of each object in the array, and contains the links describing the next object and any children that the object may have. The object tree is organized in a somewhat unconventional manner, as illustrated in the diagram below. Basically, each object in the tree contains in its header structure three links, giving the object indices of the next object, the first child object and the last child object respectively. A value of –1 in any of these fields indicates that there is no child or next object; however, as the last child of any object always uses the `ob_next` field to hold the object index of its parent, only one object in a tree will have a value of –1 in the `ob_next` field – this is the last object at the root level. Most if not all object trees have a single root object, which is always the object whose index is zero. The other fields in the object header describe what the object is and how it is displayed, and are described later.

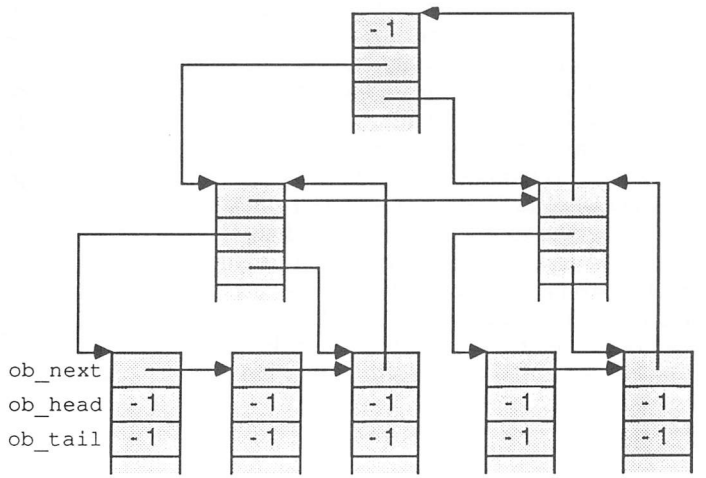


Figure 6.1 Object Tree Structure

The type *OBJECT* * is the type that appears in the parameter lists of the bindings, and the only one of these types that a simple application (or even quite a complicated one) is likely to use. If all form interaction is performed using the function `form_do` (see section 7.1), and the functions described in this section are sufficient to perform all the setting up of a form and reading of results from a form that the application requires, and all forms are created by a resource editor, then the application will never have cause to access an object header directly.

Each object in the tree is described by its object header structure. As well as the links described above, the structure contains a number of information fields as follows:

`ob_type` A two-byte integer describing what sort of object this header is describing. As there are only 13 object types, only the low order byte of this value is used by GEM AES.

`ob_flags` A bitmap containing a number of flags describing how the object is to behave in a form – these flags are set up when the object is created, and normally are not changed subsequently, as to do so would change the function of the form.

`ob_state` A bitmap containing a number of flags describing how the object is to be displayed. These flags will change as a user interacts with a form to reflect the changes in a form's appearance (such as highlighting selected objects, and so on).

`ob_spec` This is a long (4-byte) value, which contains either a pointer to a structure containing further information, or information about the object's color and border, depending upon the type of the object as specified by the `ob_type` field.

`ob_x`, `ob_y`, `ob_width`, `ob_height`
The coordinates and size of the object, in pixels. All coordinates are relative to the x and y coordinates of the object's parent (or to the screen for the root object). In a properly constructed tree, all objects must lie completely within the rectangle of their parent object.

OB_TYPE Values

The values of the `ob_type` field and the associated object types are as follows:

Value and Name	Type of Object
20 G_BOX	A box – the `ob_spec` field describes the color of the box and its border characteristics.
21 G_TEXT	A text item – the `ob_spec` field contains a pointer to an object of type *TEDINFO*.
22 G_BOXTEXT	A box containing a text item – the `ob_spec` field contains a pointer to an object of type *TEDINFO*.
23 G_IMAGE	A bit image picture – the `ob_spec` field contains a pointer to an object of type *BITBLK*.
24 G_PROGDEF	A programmer defined object – the `ob_spec` field contains a pointer to an object of type *APPLBLK*.
25 G_IBOX	An unfilled box – the `ob_spec` field describes the border thickness and color. If the border thickness is zero, it is invisible, but may be useful as a parent to group other objects, for example a set of radio buttons.
26 G_BUTTON	A text string displayed in the centre of a box, normally used for buttons – the `ob_spec` field contains a pointer to a null-terminated string.
27 G_BOXCHAR	A box containing a single character, displayed in the system font – the `ob_spec` field describes the box color and border characteristics.
28 G_STRING	A text string in the system font – the `ob_spec` field contains a pointer to a null-terminated string.
29 G_FTEXT	Formatted text – the `ob_spec` field contains a pointer to an object of type *TEDINFO*.
30 G_FBOXTEXT	A box containing formatted text – the `ob_spec` field contains a pointer to an object of type *TEDINFO*.
31 G_ICON	An icon – the `ob_spec` field contains a pointer to an object of type *ICONBLK*.
32 G_TITLE	A text string used for menu titles, otherwise identical to G_STRING.

OB_FLAGS Values

The ob_flags field in the header describes how the object behaves in a form, such as whether the object may be selected with the mouse or keyboard, and whether its selection indicates that the form processing is over. The ob_flags field will normally be set up when the tree is created, either in a resource file or by an application, and subsequently not altered, but referenced by any form handling routine. Each bit in the value corresponds to one object attribute, and may be referenced using combinations of the constants listed below (which are declared in the file AESBIND.H):

Value	Name	Function
0x0000	NONE	No attributes selected.
0x0001	SELECTABLE	The object may be selected with the mouse. Such an object will be displayed highlighted by form_do (see section 7.1) when the user clicks on it. Selectable buttons are drawn with a thicker border.
0x0002	DEFAULT	The object is to be selected when the user presses return or enter. Only one object in a tree may have this attribute set. A default button will normally have the exit flag set too, and is displayed with a blackened outline.
0x0004	EXIT	When the user selects an object with the exit attribute set, it indicates that interaction with the form is complete. The object should also be selectable. A form may have several objects with the exit attribute set.
0x0008	EDITABLE	The object may be edited by the user. Only objects of type G_FTEXT or G_FBOXTEXT can be edited. The user can select which editable text object is currently being edited using the tab, backtab, up arrow and down arrow keys.

0x0010	RBUTTON	The object is a radio button. When an object with this attribute set is selected, all its siblings (children of the same parent) with the RBUTTON attribute set will have their SELECTED bit in the `ob_state` field cleared (see below). Thus the operation is like the waveband selector on an old radio, where pressing one button in causes the others to pop out. Objects of type G_IBOX are useful as parents of sets of radio buttons.
0x0020	LASTOB	Indicates that the object is the last object in the tree (in other words it has the highest object index). This flag bit is used by GEM AES when for example determining whether there is another editable text field to move to when the user types the tab key.
0x0040	TOUCHEXIT	When the user clicks on an object with this attribute set, it indicates that interaction with the form is complete. Note that this is not quite the same as the EXIT attribute – if the user clicks on an EXIT object, `form_do` will not return until the mouse button is released, and if the mouse is moved outside the object before the button is released, the object will not be selected and `form_do` will not return. A TOUCHEXIT object will cause `form_do` to return as soon as the user presses the mouse button. The object need not be a button – some forms which simply provide information have the TOUCHEXIT bit set on all objects, so that as soon as the mouse is pressed anywhere in the form, `form_do` returns.
0x0080	HIDETREE	An object with this attribute set will not be drawn by the `objc_draw` function, or found by the `objc_find` function, or play any part in the form interaction, nor will any of its children. This is useful for concealing part of a form that is not relevant at a particular time.
0x0100	INDIRECT	If this attribute is set, it indicates that the value in the `ob_spec` field is a pointer to the value described above, rather than containing the value itself.

OB_STATE Values

The `ob_state` field in the header describes the current state of the object as a result of interaction with a form by the user, and controls how the object is displayed. Each bit in the value indicates whether a certain state is in force. The values may be referenced using combinations of the constants listed below (which are declared in the file AESBIND.H):

Value	Name	Function
0x0000	NORMAL	No object state bits selected.
0x0001	SELECTED	The object is selected, and is displayed highlighted. The application might for example set this bit on buttons representing options currently in force, then call `form_do` to allow the user to alter the settings by selecting other objects or clicking selected objects to deselect them. The application would test this bit on all relevant objects when `form_do` returns, to see what the options the user chose.
0x0002	CROSSED	The object is drawn with a cross through it. This could be used to indicate selection using a 'check box'.
0x0004	CHECKED	The object is drawn checked – with a tick mark or other selector beside it.
0x0008	DISABLED	The text of the object is drawn in gray, indicating to the user that this is not available. The SELECTABLE bit in the `ob_flags` field could also be reset to disallow selection of the object.
0x0010	OUTLINED	Under GEM version 1.1, the object is drawn with its bounding rectangle outlined. Under GEM version 2.0 the object is drawn with a drop shadow, as for the SHADOWED state bit below.
0x0020	SHADOWED	The object is drawn with a drop shadow.
0x0040	DRAW3D	This only applies to G_ICON type objects, and causes the icon mask to be drawn 3 times in a diagonal line, giving an impression of depth.
0x0080	WHITEBAK	This only applies to G_ICON type objects. When this attribute is set and the background color is white, the icon's mask and the rectangle surrounding its text will not be drawn.

OB_SPEC Values

For objects of type G_BOX, G_IBOX and G_BOXCHAR, the `ob_spec` field contains an unsigned long integer describing the object's color and border. This integer is divided into three portions, describing the color of the various parts of the box, the thickness of its border, and for G_BOXCHAR the ASCII code of the character the box contains. The border thickness is described by the low order byte of the high order word – this is the second byte in 68000 systems and the third in 8086 systems. A value of zero in the thickness field indicates no border, a positive value indicates the border is that number of pixels thick inwards from the object's rectangle, and a negative value indicates that the border is that number of pixels outwards from the object's rectangle.

The character code is contained in the high byte of the high word – this is the first byte in 68000 systems and the last in 8086 systems. For G_BOX and G_IBOX objects, this byte should contain zero.

The low order word describes the color, and is itself subdivided. The 4 most significant bits describe the border color in the range 0 to 15, the next 4 describe the text color where appropriate. The next bit selects transparent (0) or replace (1) mode – see the VDI manual for details; this is followed by 3 bits selecting the style in which the box is to be filled, in the range 0 to 7. A value of 0 means hollow fill, 7 means solid fill, and intermediate values give dither patterns of increasing intensity. The color used for these fill patterns is specified by the remaining four bits. This color word format is also used in the *TEDINFO* structure to which the `ob_spec` value may point, as described below. Note that objects of type G_IBOX are not filled, and therefore only the border color is used from this color word. When specifying values for the color word as four digit hexadecimal constants, the first digit gives the border color, the second digit the text color, the third digit the writing mode and fill style, and the last digit the inside color.

The *TEDINFO* structure

For objects of type G_TEXT, G_BOXTEXT, G_FTEXT and G_FBOXTEXT (known collectively as text objects), the ob_spec field contains a pointer to a structure of type *TEDINFO*, defined as follows:

```
typedef struct
  { char *te_ptext, *te_ptmplt, *te_pvalid;
    WORD te_font, te_resvd1, te_just, te_color,
         te_resvd2, te_thickness, te_txtlen, te_tmplen;
  } TEDINFO;
```

G_TEXT and G_BOXTEXT objects basically differ from G_STRING and G_BUTTON objects only in the additional control over the appearance that is available. For G_STRING and G_BUTTON, the ob_spec field points to the string, and there is no color or border information – the text is always displayed in black. The G_TEXT and G_BOXTEXT objects allow text to be displayed in any color, by setting the fields of the *TEDINFO* structure appropriately. The te_ptext field points to the string to be displayed – null terminated as usual – and the te_tmplt and te_pvalid fields are not used.

G_FTEXT and G_FBOXTEXT objects are known as editable text objects (and will normally have the EDITABLE bit set in the ob_flags word – see above. The te_ptext field points to the text that the user may edit – typically an application would set this field to a default or suggested text, allow the user to interact with the form using form_do as described in section 7.1, then read the value to see what the user entered. In order to simplify this process, two additional functions objc_text and objc_newtext are provided in the Prospero C bindings – see sections 6.13 and 6.14 for further details. The te_ptmplt field provides a template into which the text must be entered – any character position in this string containing an underscore indicates an editable position, while other characters are displayed as prompt or separator information. The maximum lengths of these two strings are given by the parameters te_txtlen and te_tmplen respectively. The te_pvalid string contains one character for every editable position in the template, indicating what characters may be entered in that position as follows:

9	Allow only digits 0 to 9
A	Allow upper case letters and space
a	Allow upper and lower case letters and space
N	Allow upper case letters, digits and space
n	Allow upper and lower case letters, digits and space
F	Allow all valid filename characters, and query (?), asterisk (*) and colon (:)
P	Allow all valid path name characters, and backslash (\), query (?), asterisk (*) and colon (:)
p	Allow all valid path name characters, and backslash (\) and colon (:)
X	Allow anything

If a character typed does not match the corresponding character in the `te_pvalid` string, it will be rejected, unless that character appears subsequently in the `te_ptmplt` string, in which case the cursor will be moved to the first editable position after the character, filling the `te_ptext` string with blanks up to that point. See `objc_edit` in section 6.7 for information of how to use editable text objects in forms, though this will normally be done automatically by using the function `form_do` (see section 7.1).

The remaining fields govern the way the text is displayed. The `te_font` field indicates whether the text is to be displayed in the system font (`te_font` = 3) or the small system font (`te_font` = 5). The text can be output left, right or centre aligned with values of 0, 1, or 2 in the `te_just` field respectively. The `te_color` field governs the color and pattern of the box and text in the same way as the color word in the `ob_spec` field of G_BOXCHAR items described above. The thickness of the box is given by `te_thickness`, with positive values indicating inside thickness, zero no thickness and negative values outside thickness, also as described above for the `ob_spec` value of G_BOXCHAR objects.

The *ICONBLK* structure

For objects of type G_ICON, the `ob_spec` field contains a pointer to a structure of type *ICONBLK*, defined as follows:

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

The `ib_pmask` and `ib_pdata` fields are pointers to the bit images representing the mask and data defining the icon. The bit images are arrays of WORD, and can be any multiple of 16 pixels wide, and any number of pixels high.

The `ib_ptext` field is a pointer to a null-terminated string containing the text to be displayed beneath the icon.

The `ib_char` field contains a character to be displayed on the icon – this can be used for example to give the drive letter on an icon representing a disk drive. The value is interpreted as an unsigned word, with the top 4 bits giving the foreground color, the next four the background color, and the low order byte the character to be displayed. The offset in pixels (relative to the icon position in `ib_xicon` and `ib_yicon`) of the character is given by the fields `ib_xchar` and `ib_ychar`.

The fields `ib_xicon` and `ib_yicon` describe the position of the icon, relative to the object rectangle defined in the object header, while `ib_wicon` and `ib_hicon` give its width and height in pixels. Note that the width must be divisible by 16.

The fields `ib_xtext` and `ib_ytext` describe the position of the icon's text, relative to the object rectangle defined in the object header. The text is displayed centred in a rectangle whose width and height are defined by the fields `ib_wtext` and `ib_htext`.

The field `zero` serves no purpose, and a value of 0 should be placed there.

The *BITBLK* structure

For objects of type G_IMAGE, the `ob_spec` field contains a pointer to a structure of type *BITBLK*, defined as follows:

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

The `bi_pdata` field is a pointer to the bit image that this object represents. The bit image can be any number of words wide, and any number of rows high.

The fields `bi_wb` and `bi_hl` give the width in bytes (this must be even) and height in lines of the bit image respectively. The x and y offsets of the image are given by the fields `bi_x` and `bi_y`.

The field `bi_color` describes the color to be used to display the bit image. This should be in the range 0 to 15. A value of –1 may also be specified, indicating that the image should be blitted in opaque rather than transparent mode.

The *APPLBLK* structure

For objects of type G_PROGDEF, the value in the ob_spec field points to an object of type *APPLBLK*, declared as follows:

```
typedef struct
  { void *ab_code;
    long ab_parm;
  } APPLBLK;
```

The ab_code field contains the address of a function which will be called to draw the object whenever GEM AES draws the tree. This routine may use VDI calls to draw the structure, but must not make any calls to the AES. GEM AES will pass the address of a structure of type *PARMBLK* on the stack – the structure of this is as follows:

```
typedef struct
  { OBJECT *pb_tree;     /* The tree being drawn    */
    WORD pb_obj,         /* and the object          */
         pb_prevstate,   /* The object's old state */
         pb_currstate,   /* .. and new state        */
         pb_x, pb_y,
         pb_w, pb_h,     /* Its position and size   */
         pb_xc, pb_yc,
         pb_wc, pb_hc;   /* Clipping rectangle      */
    long pb_parm;        /* Value in ab_parm field */
  } PARMBLK;
```

This type is not included in AESBIND.H, as the code to draw the object cannot be coded in Prospero C (as all registers must be preserved), and is described above simply for reference. The code must return the value of pb_currstate in a register.

The ab_parm field contains a value whose meaning may be defined by the application, which is passed to the application's code in the pb_parm field when it is called to draw the object. This might for example be used to differentiate between several different application defined objects.

6.1 Add Object to Tree `objc_add`

Add Object to Tree is used to establish a parent-child relationship between two objects in the same object tree array. Note that Prospero C provides the additional bindings `objc_create` (section 6.16) and `objc_item` (section 6.17) to assist in the setting up of object trees.

6.1.1 Definition

The Prospero C definition of Add Object to Tree is :

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

6.1.2 Purpose

This function can be used by an application to add an object to the object tree specified by the parameter `tree`. An object tree is an array of objects of type *OBJECT* – this function does not cause an element to be added to the array, but simply adjusts the `ob_next`, `ob_head` and `ob_tail` fields of the two specified elements of the array, so that one becomes the child of the other. In order to add a new item to a tree, a free element of the array must be found. This might be done by unlinking an existing element from the tree using `objc_delete` (section 6.2), or by keeping track of the first free element in the array using the LASTOB bit in the `ob_flags` field of the last object in the tree.

Having found a free element, the fields of the *OBJECT* type structure must be set up to describe the object being added – see the introduction to section 6 for further details. When the function is called, the effect is to set the `ob_tail` field of the parent, the `ob_next` field of the child and either the `ob_next` field of the previous last child of the parent (which now becomes the last but one child), or the `ob_head` field of the parent if it was childless, so that the child is incorporated into the tree at the end of the parent's list of offspring. The `ob_head` and `ob_tail` fields of the child are not altered, so that if the child was itself the root of a subtree, the entire subtree would be inserted into the parent tree. This may be used in conjunction with `objc_delete` (see section 6.2) to move an entire branch of a tree from one parent to another.

In order to allow dialog boxes to be created with the minimum of fuss, Prospero C provides some additional bindings to create object trees and add items to them – see `objc_create` (section 6.16) and `objc_item` (section 6.17).

6.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree to be updated** The pointer to the array containing the tree to be updated, which contains the parent and child object headers in the elements `tree[parent]` and `tree[child]` respectively.
`parent`	*WORD*	**Parent object index** The index within the object tree array (referred to as the object index) of the object which is to become the parent of the object specified by the parameter `child`.
`child`	*WORD*	**Child object index** The index within the object tree array of the object which is to become the child of the object specified by the parameter `parent`. This object should not already be linked into the tree.

6.1.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

6.1.5 Example

```
OBJECT * the_tree;
WORD old_object, new_parent;


  /* Move an object and children from one parent to
     another */
  objc_delete(the_tree, old_object);
  objc_add(the_tree, new_parent, old_object);
```

6.2 Delete Object from Tree `objc_delete`

Delete Object from Tree is used to remove an object from a tree by unlinking it from its parent.

6.2.1 Definition

The Prospero C definition of Delete Object from Tree is :

```
WORD objc_delete(OBJECT *tree, WORD delob);
```

6.2.2 Purpose

This function can be used by an application to delete an object from the object tree specified by the parameter `tree`. The object is deleted from the tree simply by altering the link fields of its parent and/or sibling objects in the tree – the object header information in the specified array element will not be altered, nor will the space be returned to the operating system. However the object will not be included in the effect of any subsequent `objc_draw` or `objc_find` calls, and the array element is in effect available for re-use, perhaps to set up new values in it before adding it to the tree in a different position. If an application wishes to move an object's position in a tree, other than changes which do not alter its parent but simply its position in the list of children, then it should be removed from the tree using `objc_delete` before placing it in its new position using `objc_add` (section 6.1). The `ob_head` and `ob_tail` links of the object being removed are not altered, so that if it has any children they will remain linked to it, and therefore be unlinked from the tree. In this way an entire branch of the tree can be unlinked from the tree, perhaps to be moved to a different position in the tree hierarchy.

Note that objects (and their children) can be temporarily excluded from the effects of `objc_hide` and `objc_draw` without removing them from the tree, by setting the HIDETREE bit in the `ob_flags` field.

6.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree to be updated** The pointer to the array containing the tree to be updated, which contains the object description of the object to be removed in element `tree[delob]`.
`delob`	*WORD*	**Deleted object index** The index within the object tree array (the object index) of the object which is to be deleted.

6.2.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

6.2.5 Example

```
OBJECT *the_tree;
WORD object;
.
.
  /* Remove an entire arm of an object tree */
  objc_delete(the_tree, object);
```

6.3 Draw Objects in Tree `objc_draw`

Draw Objects in Tree is used to draw any object, and optionally any of its children, contained in an object tree.

6.3.1 Definition

The Prospero C definition of Draw Objects in Tree is :

```
WORD objc_draw(OBJECT *tree, WORD drawob, WORD depth,
               WORD xc, WORD yc, WORD wc, WORD hc);
```

6.3.2 Purpose

This function can be used by an application to draw an object in the object tree specified by the parameter `tree`. The object's children may also be drawn, up to the number of generations specified by the parameter `depth`. The position on the screen where the object is drawn is determined by the `ob_x` and `ob_y` fields of the root object of the tree (object index 0), and the relative offsets in the `ob_x` and `ob_y` fields of any descendants of the root which are ancestors of the object to be drawn. This screen offset can be discovered by using the `objc_offset` function (section 6.5). The manner in which each object is drawn is determined by the relevant values in the object's header structure. Note that any object whose HIDETREE bit is set in the `ob_flags` field will not be drawn, nor will any of its descendants.

A clipping rectangle is given, outside which no screen output will occur. This might be used to clip the output of the object to the visible portion of a window – see section 11 for further details of how this might be achieved.

6.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT **	**Tree containing object to be drawn** The pointer to the object tree array containing the descriptions of the objects to be drawn (in element `tree[drawob]` and its descendants).
`drawob`	*WORD*	**Starting object index** The index within the object tree array (referred to as the object index) of the object at which drawing is to start. This will most frequently be zero (the root object), to draw the entire tree.
`depth`	*WORD*	**Number of levels to draw** The number of generations of children of the starting object which are to be drawn. Thus a value of 0 indicates that the starting object only is to be drawn, while 1 would draw its children but not their children, and so on. A suitably large value can be used to make all descendants appear – no tree is allowed more than 32767 objects, even if memory were available.
`xc` `yc`	*WORD* *WORD*	**X coordinate of clip rectangle** **Y coordinate of clip rectangle** The x and y coordinates of the top left hand corner of the rectangle to which all objects output are to be clipped.
`wc` `hc`	*WORD* *WORD*	**Width of clip rectangle** **Height of clip rectangle** The width and height (in pixels) of the rectangle to which all objects output are to be clipped.

6.3.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

6.3.5 Example

```
OBJECT * the_tree;

  /* Draw the root and its first generation children,
     clipping to the given rectangle */
  objc_draw(the_tree, 0, 1, 100, 100, 200, 50);
```

6.4 Find Object under Point `objc_find`

Find Object under Point is used to find which object in a tree lies under a specified point. This can be used to find which object has been selected with the mouse.

6.4.1 Definition

The Prospero C definition of Find Object under Point is :

```
WORD objc_find(OBJECT *tree, WORD startob, WORD depth,
               WORD mx, WORD my);
```

6.4.2 Purpose

This function can be used by an application to find which object in the object tree specified by the parameter `tree` lies under the specified point. This is normally used to discover which object a user has selected, by seeing what object lies under the mouse cursor's position when a mouse button click is detected. The depth and starting point of the search can be specified with the parameters `startob` and `depth`. Note that any object whose HIDETREE bit is set in the `ob_flags` field will not be searched, nor will any of its descendants.

6.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree to be searched** The pointer to the array containing the tree which is to be searched.
`startob`	*WORD*	**Starting object index** The index within the object tree array (referred to as the object index) of the object at which the search is to start.
`depth`	*WORD*	**Number of levels to search** The number of generations of children of the starting object which are to be searched. Thus a value of 0 indicates that the starting object only is to be checked, while 1 would also check its children but not their children, and so on. A suitably large value can be used to make all descendants be included – no tree is allowed more than 32767 objects, even if memory were available.
`mx` `my`	*WORD* *WORD*	**X coordinate of search point** **Y coordinate of search point** The x and y coordinates of the point under which an object is to be found.

6.4.4 Function Result

The function returns the object index of the object found, or –1 if no object searched lies under the specified point. The search is done on a depth first basis – if a child is under the specified point, so must its parent be, but it is the index of the child which is returned.

6.4.5 Example

```
OBJECT *the_tree;
WORD mx, my, screen_width, screen_height;
WORD dummy;
WORD selected;

  /* Set up tree containing icons and other goodies */

  /* Draw the entire tree */
  objc_draw(the_tree, 0, 32767, 0, 0, screen_width,
            screen_height);

  /* Wait for a click on left button */
  evnt_button(1, 1, 1, &mx, &my, &dummy, &dummy);

  selected = objc_find(the_tree, 0, 32767, mx, my);
  if (selected == -1)
    { /* click wasn't on an object
         - could allow a group selection */
      ...
    }
  else
    { /* Deal with selected object */
      ...
    }
```

6.5 Calculate Object Offset `objc_offset`

Calculate Object Offset is used to find the screen coordinates of a particular object in an object tree.

6.5.1 Definition

The Prospero C definition of Calculate Object Offset is :

```
WORD objc_offset(OBJECT *tree, WORD obj,
                 WORD *poffx, WORD *poffy);
```

6.5.2 Purpose

This function can be used by an application to return the screen coordinates of the top left hand corner of an object in an object tree. The values in an object's `ob_x` and `ob_y` fields are relative to the position of its parent, so that the offset of an object relative to the screen can only be determined by adding together the offsets of all its ancestors, including the root, whose offsets are relative to the screen. This is what this function does.

6.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT* *	**Tree to be searched** The pointer to the array containing the tree concerned.
`obj`	*WORD*	**Object index** The index within the object tree array (referred to as the object index) of the object whose screen offset is to be calculated.
`poffx` `poffy`	*WORD* * *WORD* *	**X offset of object** **Y offset of object** These parameters point to objects which return the x and y offsets of the specified object in pixels relative to the screen origin (the top left hand corner).

6.5.4 Function Result

The value returned will be zero if an error occurred, or greater than zero to indicate no error was detected.

6.5.5 Example

```
OBJECT * the_tree;
WORD x, y, my_obj;

  /* Find offset of my_obj */
  objc_offset(the_tree, my_obj, &x, &y);
```

6.6 Alter Object Order `objc_order`

Alter Object Order is used to alter the order of the children of a particular parent object.

6.6.1 Definition

The Prospero C definition of Alter Object Order is :

```
WORD objc_order(OBJECT *tree,
                WORD mov_obj, WORD newpos);
```

6.6.2 Purpose

This function can be used by an application to move an object to a new position in its parent's list of children. To make any alteration to the object tree which involves a change of parent, `objc_delete` followed by `objc_add` should be used, as described in sections 6.1 and 6.2. The order of children will affect the order in which they are drawn or searched by the `objc_draw` (section 6.3) and `objc_find` (section 6.4) functions, which may be relevant if the objects concerned overlap.

6.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree to be searched** The pointer to the array containing the tree concerned.
`mov_obj`	*WORD*	**Object index of child to move** The object index within the tree of the object whose position in its parent's list of children is to be changed.

`newpos` *WORD* **New position of child**

The new position in the parent's list of children which the object is to take, as follows :-

0 – last child
1 – last but one child
2 – last but two (etc.)
–1 – first child

6.6.4 Function Result

The value returned will be zero if an error occurred, or greater than zero to indicate no error was detected.

6.6.5 Example

```
OBJECT *the_tree;
WORD child_obj;

  /* Make object the first child */
  objc_order(the_tree, child_obj, -1);
```

6.7 Edit Text Object `objc_edit`

Edit Text Object is used to allow a user to interact with an editable text object on a form or dialog.

6.7.1 Definition

The Prospero C definition of Edit Text Object is :

```
WORD objc_edit(OBJECT *tree, WORD obj, WORD inkey,
               WORD *idx, WORD kind);
```

6.7.2 Purpose

This function can be used by an application to allow a user to interact with an editable text item on a form or dialog, which are really just special forms of object trees. Most applications will not need to use this function, even if they make use of editable text fields, as all such interactions with a dialog are handled internally by the `form_do` function (section 7.1). Only if an application needs to use forms with features not supported by `form_do` – perhaps slider bars for setting values, or the ability to perform background calculations while awaiting the user's response – will it need to drive the editable text handling itself.

To handle interaction with a form, an application will have to keep track of which item is being processed (GEM version 2.0 provides two routines to help here – `form_keybd` (section 7.6) and `form_button` (section 7.7) – but these are not available in GEM version 1.1), wait for events and process them when they arrive. When a keystroke is detected, the application should first check whether it is a tab, backtab, uparrow, downarrow or return character, which might change to a new text item (this can be done using `form_keybd` in GEM version 2.0), then process the character against the current editable text object using this function.

The function performs 3 separate functions, according to the value of the parameter `kind`:

Value	Name	Function
0	ED_START	Reserved for future use.
1	ED_INIT	Turn on text cursor on object, and combine text and template into a formatted string. The value of `inchar` is not relevant, nor the initial value of `idx`. On return, `idx` will contain the initial cursor position within the string (immediately after the last character). This should be used when the object becomes the current editable text object.
2	ED_CHAR	Validate the key pressed against the template, update the string and the cursor position. This is used when a key press is detected.
3	ED_END	Turn off text cursor. Use this when the object is no longer the current object, or when form processing is about to finish.

The above macros are provided in the AESBIND.H header file.

The function definition in Digital Research's GEM version 1.1's C bindings is slightly different from that in GEM version 2.0 (in GEM version 1.1, the new character position is returned in a separate WORD * parameter), though they are functionally equivalent and the underlying GEM function has not changed. The Prospero C binding provided is the same as the GEM version 2.0 version of the Digital Research bindings, which is slightly easier to use.

6.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT* *	**Tree containing object** The pointer to the array containing the tree describing the form which is being processed.

`obj` *WORD* **Object index of editable text**

The index within the object tree array (referred to as the object index) of the editable text object. This must have an object type of G_FTEXT or G_FBOXTEXT in the `ob_type` field.

`inkey` *WORD* **Key to be validated**

The character which the user has typed, which is to be inserted into the text at the current text position, if it matches the corresponding validation character in the template. The value passed should be the same as that returned by `evnt_multi` (section 4.6) or `evnt_keybd` (section 4.1) when the key press was detected.

`idx` *WORD ** **Cursor position in raw text**

Points to an object containing the position within the raw text at which the character is to be inserted if possible. The value in this object is updated, ready to be passed to the function again when the next character is received. In the GEM 1.1 Digital Research bindings the updated value was returned via a separate parameter called `ob_ednewidx`.

`kind` *WORD* **Operation required**

A value in the range 0 to 3 indicating which function is required, as described above in 6.7.2.

6.7.4 Function Result

The value returned will be zero if an error occurred, or greater than zero to indicate no error was detected.

6.7.5 Example

```
/* A very simple form handler, with no support for mouse
   selection of which field to edit, and very much dependent
   on all the editable text objects in the form having
   consecutive numbers.
   If form_keybd (section 7.6) were used to process the tab,
   backtab etc. cases, this restriction could be easily
   removed, as shown in the example in section 7.6.5 */

#define text1obj 20 /* Provided by resource editor when form */
#define text2obj 21 /* was created */
#define text3obj 22

OBJECT *the_form;
WORD current_obj, key, pos;

  current_obj = text1obj;
  objc_edit(the_form, current_obj, 0, &pos, ED_INIT);
          /* Initialize first editable text field */
  do
  { key = evnt_keybd();
    switch (key)
    { case tab:
      case cursordown:
        if (current_obj != text3obj)
        { objc_edit(the_form, current_obj, key, &pos, ED_END);
                  /* Remove cursor from current text field */
          current_obj ++;
          objc_edit(the_form, current_obj, key, &pos, ED_INIT);
                  /* Initialize new text field */
        }
        break;

      case backtab:
      case cursorup:
        if (current_obj != text1obj)
        { objc_edit(the_form, current_obj, key, &pos, ED_END);
          current_obj --;
          objc_edit(the_form, current_obj, key, &pos, ED_INIT);
        }
        break;

      case carriage_return:
        objc_edit(the_form, current_obj, key, &pos, ED_END);
                  /* Remove cursor ready to terminate */
        break;

      default:
        objc_edit(the_form, current_obj, key, &pos, ED_CHAR);
                /* Process the character typed */
        break;
    } /* switch */
  } while (key != carriage_return);
```

6.8 Change Object State `objc_change`

Change Object State is used to set the value of the `ob_state` field of an object. It will often be simpler and always faster to change it directly, by a statement such as

```
tree[index].ob_state = newstate;
```

or using the additional Prospero C binding `objc_newstate` (section 6.10).

6.8.1 Definition

The Prospero C definition of Change Object State is :

```
WORD objc_change(OBJECT *tree, WORD drawob, WORD depth,
                 WORD xc, WORD yc, WORD wc, WORD hc,
                 WORD newstate, WORD redraw);
```

6.8.2 Purpose

This function can be used by an application to change the value in an object's `ob_state` field, which determines the way in which it is drawn – see the introduction to section 6 for further details of the `ob_state` field. It offers two features not available when simply altering the state field as described above: the object may optionally be redrawn using the new state, and a clipping rectangle may be specified. The GEM AES documentation states that only those parts of the object which are contained within the clipping rectangle will have their state changed – it is hard to see what this means, as the object is a single entity with a single state field. The clipping rectangle seems more relevant to the (optional) redraw.

There is a parameter called `depth`, currently reserved, which presumably is intended to allow the new state to be applied to all children of the specified object up to a particular number of levels in some future release. When this is implemented, the clipping rectangle may be of more relevance! Applications should ensure that they pass a value of zero in the depth field, otherwise they may find that the effect of the call suddenly changes dramatically in future versions of GEM.

6.8.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
tree	*OBJECT**	**Tree containing object** The pointer to the array containing the tree describing the form which is being processed.
drawob	*WORD*	**Object to be changed** The index within the object tree array (referred to as the object index) of the object whose state is to be changed.
depth	*WORD*	**Reserved** Reserved for future use. A value of zero must be passed.
xc yc wc hc	*WORD* *WORD* *WORD* *WORD*	**Clipping rectangle X coordinate** **Clipping rectangle Y coordinate** **Clipping rectangle width** **Clipping rectangle height** The clipping rectangle to be used in the optional redraw. If the specified object does not lie within the clipping rectangle, it may not be changed.
newstate	*WORD*	**New object state** The new value for the specified object's ob_state field.
redraw	*WORD*	**Redraw flag** If this is one, the object will be redrawn in its new state using the specified clipping rectangle. If zero, no redraw will occur.

6.8.4 Function Result

The value returned will be zero if an error occurred, or greater than zero to indicate no error was detected.

6.8.5 Example

```
#define NORMAL 0        /* In AESBIND.H */

OBJECT *form;
WORD end_object;

  /* Interact with a form */
  end_object = form_do(form, 0);

  /* The object which caused exit from the form has its
     selected state bit set - must reset this before
     reusing the form */

  objc_change(form, end_object, 0, 0, 0, 500, 500,
              NORMAL, 0);

  /* Could have written
    form[end_object].ob_state = NORMAL; */
```

6.9 Return Object State `objc_state`

Return Object State is used to return the value of the specified object's `ob_state` field. It is not part of the original Digital Research bindings, but is provided as a macro in Prospero C to simplify the access of this field. The equivalent variable access is

```
state = tree[object].ob_state;
```

6.9.1 Definition

The Prospero C definition of Return Object State is :

```
WORD objc_state(OBJECT *tree, WORD object);
```

6.9.2 Purpose

This function can be used by an application to return the value in an object's `ob_state` field, which determines the way in which it is drawn – see the introduction to section 6 for further details of the `ob_state` field. It is not provided in the original Digital Research bindings. Note that it is in fact implemented as a macro in AESBIND.H.

6.9.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index within the object tree array (referred to as the object index) of the object whose state is to be returned.

6.9.4 Function Result

The result returned is a WORD giving the current value of the object's `ob_state` field.

6.9.5 Example

```
#define button1 21          /* Sample constants from */
#define button2 22          /* resource editor */
#define button3 23
#define SELECTED 0x0001     /* From AESBIND.H file */

OBJECT *form;
WORD end_object, i;

  /* Interact with a form */
  end_object = form_do(form, 0);

  for (i = button1; i <= button3; i++)
    if (objc_state(form, i) & SELECTED)
      { /* The option corresponding to each button
           selected should be put into force */
        ...
      }
```

6.10 Set Object State objc_newstate

Set Object State is used to set the value of the specified object's ob_state field. It is not part of the original Digital Research bindings, but is provided in Prospero C to simplify the access of this field, as a complementary function to objc_state (see section 6.9). The equivalent variable access is

```
tree[object].ob_state = newstate;
```

6.10.1 Definition

The Prospero C definition of Set Object State is :

```
void objc_newstate(OBJECT *tree,
                   WORD object, WORD newstate);
```

6.10.2 Purpose

This function can be used by an application to set the value in an object's ob_state field, which determines the way in which it is drawn – see the introduction to section 6 for further details of the ob_state field. This function is not provided in the original Digital Research bindings. However, the function objc_change (see section 6.8) may be used for this purpose – this includes the ability to specify a clipping rectangle and optionally redraw the object. When the object concerned is not displayed, it is simpler and faster to use the function objc_newstate described here. Note that this function is declared as a macro in AESBIND.H.

6.10.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index within the object tree array (referred to as the object index) of the object whose state is to be set.
`newstate`	*WORD*	**New object state** The value to which the object's `ob_state` field is to be set.

6.10.4 Function Result

There is no function result.

6.10.5 Example

```
#define button1 21          /* Sample constants from */
#define button2 22          /* resource editor */
#define button3 23
#define SELECTED 0x0001     /* From AESBIND.H file */

OBJECT *form;
WORD i;

  /* Unselect all buttons before using form */
  for (i = button1; i <= button3; i++)
    objc_newstate(form, i, objc_state(form, i) &
                           (~SELECTED));
```

6.11 Return Object Flags `objc_flags`

Return Object Flags is used to return the value of the specified object's `ob_flags` field. It is not part of the original Digital Research bindings, but is provided in Prospero C to simplify the access of this field. The equivalent variable access is

```
flags = tree[object].ob_flags;
```

6.11.1 Definition

The Prospero C definition of Return Object Flags is :

```
WORD objc_flags(OBJECT *tree, WORD object);
```

6.11.2 Purpose

This function can be used by an application to return the value in an object's `ob_flags` field, which determines the way in which it is affected by mouse and keyboard input from the user when interacting with a form – see the introduction to section 6 for further details of the `ob_flags` field. It is not provided in the original Digital Research bindings. Note that this is declared as a macro in AESBIND.H.

6.11.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index within the object tree array (referred to as the object index) of the object whose flags are to be returned.

6.11.4 Function Result

The result returned is a WORD giving the current value of the object's `ob_flags` field.

6.11.5 Example

```
#define DEFAULT 0x0002          /* From AESBIND.H file */

OBJECT *form;
WORD i;

  /* Find the object whose DEFAULT bit is set. In
     practice, the tree would be scanned by following
     the pointers, to allow the detection of the case
     where no object has the default bit set */

  i = 0;
  while ((objc_flags(form,i) & DEFAULT) == 0)
    i ++;
```

6.12 Set Object Flags `objc_newflags`

Set Object Flags is used to set the value of the specified object's `ob_flags` field. It is not part of the original Digital Research bindings, but is provided in Prospero C to simplify the access of this field, as a complementary function to `objc_flags` (section 6.11). The equivalent variable access is

```
tree[object].ob_flags = newflags;
```

6.12.1 Definition

The Prospero C definition of Set Object Flags is :

```
void objc_newflags(OBJECT *tree,
                   WORD object, WORD newflags);
```

6.12.2 Purpose

This function can be used by an application to set the value in an object's `ob_flags` field, which determines the way in which it is affected by mouse and keyboard actions when a form is being processed – see the introduction to section 6 for further details of the `ob_flags` field. It is not normal practice to alter the flags of an object once they have been set up – however it is sometimes useful to set the HIDETREE flag to conceal part of a form in a situation when that part is not relevant. This function is not provided in the original Digital Research bindings. Note that this function is declared as a macro in AESBIND.H.

6.12.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index within the object tree array (referred to as the object index) of the object whose flags are to be set.
`newflags`	*WORD*	**New object flags** The value to which the object's `ob_flags` field is to be set.

6.12.4 Function Result

There is no function result.

6.12.5 Example

```
#define subtree 15          /* Sample constant from
                               resource editor */
#define HIDETREE 0x0080     /* From AESBIND.H file */

OBJECT *form;

  objc_newflags(form, subtree,
                (objc_flags(form, subtree) | HIDETREE));

  /* Hide the object and its children before displaying
    and using form */
```

6.13 Return Object Text `objc_text`

Return Object Text is used to return the text associated with a given object. It is not part of the original Digital Research bindings, but is provided in Prospero C to simplify the access of this field.

6.13.1 Definition

The Prospero C definition of Return Object Text is :

```
char *objc_text(OBJECT *tree, WORD object);
```

6.13.2 Purpose

This function can be used by an application to return a pointer to the text associated with an object. It would normally only be used for editable text objects, to discover the text which the user entered after a user has interacted with a form.

This function is not provided in the original Digital Research bindings.

6.13.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The object index of the object whose text is to be returned.

6.13.4 Function Result

A pointer to the text associated with the specified object is returned. If the specified object is not of type G_STRING, G_BUTTON, G_TITLE, G_TEXT, G_BOXTEXT, G_FTEXT or G_FBOXTEXT, the value returned will be `NULL`.

6.13.5 Example

```
#define textobj 20    /* Sample constant from
                         resource editor */

OBJECT *form;
char *result;

  /* Specify first object to edit */
  form_do(form, textobj);

  /* get pointer to the string the user typed */
  result = objc_text(form, textobj);
```

6.14 Set Object Text `objc_newtext`

Set Object Text is used to initialize or modify the text of any object which has a text field. For editable text objects of type G_FTEXT and G_FBOXTEXT, this can be used to set up the initial contents before allowing the user to modify it using `form_do` (section 7.1). This function is not part of the original Digital Research bindings, but is provided in Prospero C to simplify the creation and use of dialog boxes.

6.14.1 Definition

The Prospero C definition of Set Object Text is :

```
void objc_newtext(OBJECT *tree,
                  WORD object, const char *newtext);
```

6.14.2 Purpose

This function can be used by an application to specify the text of an object of type G_STRING, G_BUTTON, G_TEXT, G_BOXTEXT, G_FTEXT or G_BOXTEXT. For editable text objects, this function is useful for giving the suggested or default value of a text field that the user may alter. Other text objects are not normally altered once set up, and this function will usually only be used when a form is first created. The new text is copied to the address previously occupied by the old text, or into newly allocated memory if the text pointer was previously `NULL` – this will only be true if the object has been created using `objc_item` and not yet initialized. Thus the length of the new string must not exceed the length of the first string which was allocated to that particular object.

This function is not provided in the original Digital Research bindings.

6.14.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index of the object whose text is to be set.
`newtext`	*const char**	**Object's text field** This parameter points to a null-terminated string containing the text which will be displayed next time the object is drawn. If the specified object is of a type which has no text, this function will have no effect. Care should be taken not to exceed the space available, which depends upon the length of the first string associated with the object.

6.14.4 Function Result

There is no function result.

6.14.5 Example

```
#define textobj 20
    /* Sample constant from resource editor */

OBJECT *form;
char *result;

  /* Set up initial value */
  objc_newtext(form, textobj, "100");

  /* Draw and process the form - see section 7 */

  /* Obtain final value */
  result = objc_text(form, textobj);
```

6.15 Read/Write Object Header

objc_read
objc_write

Read and Write Object Header are provided to give easy access to the various fields which describe objects in AES object trees. Although in C these fields can be accessed quite easily using a variable access such as `tree_ptr[index].fieldname`, these bindings are still useful, for cases where a number of entire object headers are being set up (such as when creating an object tree). These functions are not part of the original Digital Research bindings.

6.15.1 Definition

The Prospero C definitions of Read and Write Object Header are :

```
void objc_read(OBJECT *tree, WORD object,
               WORD *next, WORD *head, WORD *tail,
               WORD *type, WORD *flags, WORD *state,
               unsigned long *spec,
               WORD *x, WORD *y,
               WORD *width, WORD *height);

void objc_write(OBJECT *tree, WORD object,
                WORD next, WORD head, WORD tail,
                WORD type, WORD flags, WORD state,
                unsigned long spec,
                WORD x, WORD y,
                WORD width, WORD height);
```

6.15.2 Purpose

These functions can be used to examine, alter or initialize the contents of an object header structure. See the introduction to section 6 for more information about this structure. Note that simpler functions are provided to examine or alter the `flags` or `state` fields – see sections 6.9 to 6.13.

The parameters `next` to `height` correspond to the fields of the object header described in the introduction to section 6. The function `objc_read` transfers the contents of the specified object header into the variables pointed to by the parameters, while `objc_write` transfers the values passed into the object header.

These functions are not provided in the original Digital Research bindings.

6.15.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT **	**Tree containing object** The pointer to the array containing the tree concerned.
`object`	*WORD*	**Object** The index within the object tree array of the object whose header is to be read or written.
`next`	*WORD* *WORD **	**Link to next object** This field of the object header contains the object index of the next object in the tree. This will normally the next child of the object's parent, or the parent itself for the last child. A value of –1 indicates there is no next object.
`head`	*WORD* *WORD **	**Link to first child** This field of the object header contains the object index of the first child of the object, or –1 if the object has no children.
`tail`	*WORD* *WORD **	**Link to last child** This field of the object header contains the object index of the last child of the object, or –1 if the object has no children.
`type`	*WORD* *WORD **	**Type of object** This field contains the object type. See the introduction to section 6 for a description of the types available.

`flags` *WORD* / *WORD ** **Object's flags field**

This field contains information which determines the way in which the object interacts with the mouse if used in a form. Each bit of the value has an individual meaning – these are described in the introduction to section 6.

`state` *WORD* / *WORD ** **Object's state field**

This field contains information which determines the way in which the object is drawn, and will be altered to cause changes in appearance if the object interacts with the mouse in a form. Each bit of the value has an individual meaning – these are described in the introduction to section 6.

`spec` *unsigned long* / *unsigned long ** **Object's specification**

This field is four bytes whose meaning depend upon the type of the object specified by the `ob_type` field. For simple objects, this contains information about the color, border, and fill style. For more complicated objects, this field contains a pointer to an area of memory giving further information about the object. See the introduction to section 6 for further details.

`x` *WORD* **Object's X coordinate**
`y` *or* **Object's Y coordinate**
`width` *WORD ** **Object's width**
`height` **Object's height**

These fields of the object header describe the position and size of the object's bounding rectangle. For all objects other than the root, the coordinates are relative to the top left hand corner of the parent. Note that all children of an object should lie completely within the parent's bounding box.

6.15.4 Function Result

There is no function result

6.15.5 Example

```
#define G_BUTTON 26
#define G_STRING 28
#define NORMAL   0   /*Constants from AESBIND.H file*/

OBJECT *tree;
WORD object, ob_next, ob_head, ob_tail;
WORD ob_type, ob_flags, ob_state;
unsigned long ob_spec;
WORD ob_x, ob_y, ob_w, ob_h;

/* A very inefficient way to alter an ob_type field */


objc_read(tree, object, &ob_next, &ob_head, &ob_tail,
          &ob_type, &ob_flags, &ob_state,
          &ob_spec, &ob_x, &ob_y, &ob_w, &ob_h);

if (ob_type == G_BUTTON)
  objc_write(tree, object, ob_next, ob_head, ob_tail,
             G_STRING, ob_flags, NORMAL,
             ob_spec, ob_x, ob_y, ob_w, ob_h);

/* Could have written
  if (tree[object].ob_type == G_BUTTON)
    { tree[object].ob_type = G_STRING;
      tree[object].ob_state = NORMAL; }      */
```

6.16 Create Object Tree `objc_create`

Create Object Tree is used to allocate space for an object tree so that it can be used for a dialog box, and to initialize the root of the tree. Items can then be added to the tree using `objc_add` (section 6.1) or more easily using the additional Prospero C binding `objc_item` (section 6.17). It is normally preferable to create all forms and dialogs using a resource editor, but for simple applications or where no resource editor is available, these routines can be used to set up dialogs for obtaining input from the user. This function is not part of the original Digital Research bindings.

6.16.1 Definition

The Prospero C definition of Create Object Tree is :

```
OBJECT *objc_create(WORD items,
                    WORD x, WORD y, WORD w, WORD h);
```

6.16.2 Purpose

This function is used when dynamically creating dialog forms (as opposed to reading them in from a resource file). Section 6.1 outlines how to create such a form using `objc_add` – the functions `objc_create` and `objc_item` (section 6.17) were added to the bindings by Prospero Software to simplify the function.

The form is created, and a pointer to it obtained by calling `objc_create`, specifying the number of objects which will be placed in the form. It is a good idea to allow for a few extra items in case more objects are added in later versions of the program. Each object in the form is then added individually using `objc_item` (section 6.17), specifying its parent, type, position, flags, state and color. For any object other than a box, the object will require some additional initialization. For objects which involve a string of text, the text may be set up using `objc_newtext` (section 6.14), or `objc_tedinfo` (section 6.18) for editable text objects. Other objects may need the structure pointed to by the `ob_spec` field to be set up.

This function is not provided in the original Digital Research bindings.

6.16.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
items	*WORD*	**Number of objects to be added** The number of objects which will be added to the tree. If more are added, other data will be overwritten, and unpredictable results will occur, so it is a good idea to err on the generous side.
x	*WORD*	**Root object's X coordinate**
y	*WORD*	**Root object's Y coordinate**
w	*WORD*	**Root object's width**
h	*WORD*	**Root object's height** The coordinates of the root box which is to contain the dialog form. The above coordinates are given in CHARACTERS not pixels, so that programs can easily be made independent of screen resolution.

6.16.4 Function Result

The function returns a pointer to the newly created tree, which can then be used as a parameter to `objc_item` or any of the other functions in the object and form libraries.

6.16.5 Example

```
OBJECT *my_form;
WORD cancel, ok, the_string;
WORD x, y, w, h;

  my_form = objc_create(10, 0, 0, 30, 20);
  /* Note x and y coords don't matter if form_center
    is used */

  ok = objc_item(my_form, 0, G_BUTTON,
                 SELECTABLE | EXIT | DEFAULT,
                 NORMAL,
                 22, 18, 6, 1,
                 0, 0);
  objc_newtext(my_form, ok, "OK");
  cancel = objc_item(my_form, 0, G_BUTTON,
                     SELECTABLE | EXIT,
                     NORMAL,
                     14, 18, 6, 1,
                     0, 0);
  objc_newtext(my_form, cancel, "Cancel");

  the_string = objc_item(my_form, 0, G_STRING,
                         NONE, NORMAL,
                         5, 10, 25, 1,
                         0, 1);
  obcj_newtext(my_form, the_string,
               "A piece of black text");

  form_center(my_form, &x, &y, &w, &h);
  form_dial(FMD_START, 0,0,0,0, x  y, w ,h);
  objc_draw(my_form, 0, 2, x, y, w, h);
  if (form_do(my_form, 0) == ok)
    { /* clicked ok button */
    }
  else
    { /* must have clicked cancel */
    }
```

6.17 Insert Item into Object Tree `objc_item`

Insert Item into Object Tree is used to add objects to a dialog form created using `objc_create` (section 6.16). It is not suitable for dialog forms loaded from a resource file, as these will not have free object headers allocated. This function is not part of the original Digital Research bindings.

6.17.1 Definition

The Prospero C definition of Insert Item into Object Tree is :

```
WORD objc_item(OBJECT *form, WORD parent,
               WORD otype, WORD oflags, WORD ostate,
               WORD x, WORD y, WORD w, WORD h,
               WORD border, WORD color);
```

6.17.2 Purpose

This function is used when dynamically creating dialog forms (as opposed to reading them in from a resource file). Section 6.1 outlines how to create such a form using `objc_add` – the functions `objc_item` and `objc_create` (section 6.16) were added to the bindings by Prospero Software to simplify the function.

Having created a form using `objc_create` (section 6.16), items can be added to it with this function. This causes the next free object header to be linked into the tree in the specified position, and its index returned so that the object can be referred to in subsequent calls to the object or form library routines. The fields of the object header are all set up from the information given in the parameters – however for objects of types G_ICON, G_IMAGE, G_PROGDEF, and any object containing text, the `ob_spec` field will contain a pointer to some additional information, which must be set up by the program before using the form. For G_STRING, G_BUTTON, G_TEXT and G_BOXTEXT objects, this can be done with a call to `objc_newtext` (section 6.14). G_FTEXT and G_FBOXTEXT items can be set up using the additional Prospero C binding `objc_tedinfo` (section 6.18). For objects of type G_IMAGE, G_ICON and G_PROGDEF, the additional structure must be set up explicitly (though the space for it and the pointer to it will have been allocated by `objc_item`).

This function is not provided in the original Digital Research bindings.

6.17.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
form	*OBJECT**	**Dialog form being created** The tree pointer of the tree to which the object is to be added. This should have been created using objc_create (section 6.16) – forms loaded from a resource file will not have any free space at the end of the tree, and so would cause other data to be overwritten if used.
parent	*WORD*	**Parent of object to be added** The object which is to be the parent of the new object. For simple forms, the root object (index zero) will be the parent of all other objects in the tree. However, it is sometimes useful to make a group of buttons children of an IBOX object, so that they can all be drawn, hidden or searched at a time. If radio buttons are used, all buttons which have the same parent will interact, so that selecting one causes the others to be deselected.
otype	*WORD*	**Type of object to be added** The type of the object. This should be one of the constants described at the beginning of section 6, which are declared in the header file AESBIND.H. If the object type is one where the ob_spec field contains a pointer to a structure of additional information, the space for this structure will be allocated.
oflags ostate	*WORD* *WORD*	**Object flags** **Object state** These parameters are copied to the ob_flags and ob_state fields of the new object. The introduction to section 6 describes the meanings of these two fields.

x	*WORD*	**Object's X coordinate**
y	*WORD*	**Object's Y coordinate**
w	*WORD*	**Object's width**
h	*WORD*	**Object's height**

The coordinates of the object, relative to its parent. These are given in CHARACTERS, so that programs can easily be made independent of screen resolution.

`border` *WORD* **Object border thickness**

The thickness of the border to be drawn around the object, in pixels. Positive values indicate a border drawn inwards from the edge of the object, while negative thicknesses are drawn outwards. Note that this parameter is not used for objects of types G_BUTTON, G_STRING, G_ICON, G_IMAGE or G_PROGDEF.

`color` *WORD* **Object color word**

The color of the object's border, text and fill. The format of this word is described in detail in the introduction to section 6 – basically the top four bits give the border color, the next 4 describe the text color where appropriate, the next bit selects transparent (0) or replace (1) mode, followed by 3 bits selecting the style in which the box is to be filled, then 4 bits giving the fill color. Note that this parameter is not used for objects of types G_BUTTON, G_STRING, G_ICON, G_IMAGE or G_PROGDEF.

6.17.4 Function Result

The function returns the index of the object added, which can then be used as a parameter to any of the other functions in the object and form libraries.

6.17.5 Example

```
#include <AESBIND.H>

OBJECT *my_form;
WORD cancel, ok, find_title, find_text,
     find_forwards, find_backwards, rbox;
WORD x, y, w, h;

  my_form = objc_create(10, 0, 0, 30, 20);

  /* First give the form a title ... */
  find_title = objc_item(my_form, 0, G_STRING,
                         NONE, NORMAL,
                         13, 2, 4, 1,
                         0, 0);
  objc_newtext(my_form, find_title, "FIND");

  /* Add an editable text field for search text */
  find_text = objc_item(my_form, 0, G_FTEXT,
                        EDITABLE, NORMAL,
                        3, 4, 24, 1,
                        0, 0x0100);
  objc_tedinfo(my_form, find_text,
               "@2345678901234567",
               "Find :- _________________",
               "XXXXXXXXXXXXXXXXX",
               3, 0);

  /* Make invisible box to group the radio buttons */
  rbox = objc_item(my_form, 0, G_IBOX,
                   NONE, NORMAL,
                   3, 6, 24, 2, 0, 0);

  /* ... and put two radio buttons in it */
  find_forwards = objc_item(my_form, rbox, G_BUTTON,
                            SELECTABLE | RBUTTON,
                            NORMAL,
                            2, 1, 9, 1,
                            0, 0);
  objc_newtext(my_form, find_forwards, "Forwards");
  find_backwards = objc_item(my_form, rbox, G_BUTTON,
                             SELECTABLE | RBUTTON,
                             NORMAL,
                             13, 11, 9, 1,
                             0, 0);
  objc_newtext(my_form, find_backwards, "Backwards");
```

```
/* Add a pair of exit buttons, OK and CANCEL */
ok = objc_item(my_form, 0, G_BUTTON,
               SELECTABLE | EXIT | DEFAULT,
               NORMAL,
               22, 18, 6, 1,
               0, 0);
objc_newtext(my_form, ok, "OK");
cancel = objc_item(my_form, 0, G_BUTTON,
                   SELECTABLE | EXIT,
                   NORMAL,
                   14, 18, 6, 1,
                   0, 0);
objc_newtext(my_form, cancel, "CANCEL");

/* The form is now ready to use */

form_center(my_form, &x, &y, &w, &h);
form_dial(FMD_START, 0,0,0,0, x, y, w ,h);
objc_draw(my_form, 0, 2, x, y, w, h);
if (form_do(my_form, find_text) == ok)
  { /* clicked ok button */
  }
else
  { /* must have clicked cancel */
  }
```

6.18 Initialize Editable Text Object `objc_tedinfo`

Initialize Editable Text Object is used to set up the *TEDINFO* structures describing objects of type G_FTEXT or G_FBOXTEXT added to a dialog form created using `objc_create` (section 6.16). This function is not part of the original Digital Research bindings.

6.18.1 Definition

The Prospero C definition of Initialize Editable Text Object is :

```
void objc_tedinfo(OBJECT *form, WORD object,
                  const char *ptext,
                  const char *template,
                  const char *valid
                  WORD font, WORD justify);
```

6.18.2 Purpose

This function is used after adding an object of type G_FTEXT or G_FBOXTEXT to a dialog form. These object types are editable text fields which can be filled by the user. The objects are described by an additional structure of type *TEDINFO*, whose address is contained in the object header's `ob_spec` field – these structures are described in the introduction to section 6. When `objc_item` (section 6.17) is used to add one of these objects, memory is allocated for the *TEDINFO* structure, and the `te_color` and `te_thickness` fields set up from the parameters given. However before the form can be used the contents of the other fields in the *TEDINFO* structure must be set up – this can be achieved using this function.

This function is not provided in the original Digital Research bindings.

6.18.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`form`	*OBJECT* *	**Dialog form being created** The tree pointer of the tree concerned.
`object`	*WORD*	**Editable text object to be set up** The index of the object to be initialized, as returned by `objc_item` when the object was added to the form.
`ptext`	*const char* *	**Initial text string** The initial contents of the editable text object. This is the portion which the user can change when interacting with the form. If the object is to be initially empty, an empty string `""` should be passed.
`template`	*const char* *	**Text template** The template into which the text is to be entered. This should contain an underscore character for every editable position – see the introduction to section 6 for more details.
`valid`	*const char* *	**Validation string** A string indicating which characters will be acceptable in each editable position of the template – see the introduction to section 6 for more details.
`font`	*WORD*	**Required font** This parameter is copied into the `te_font` field, which indicates whether the text is to be displayed in the small font used for icons (`font` = 5) or in the standard system font (`font` = 3).

`justify` *WORD*

Required justification

This parameter is copied into the `te_just` field, which indicates whether the text is to be displayed left justified (`justify` = 0), right justified (`justify` = 1) or centred (`justify` = 2). Centre justification of editable text object doesn't appear to work correctly under some versions of GEM.

6.18.4 Function Result

There is no function result.

6.18.5 Example

See section 6.17.5.

7 FORM LIBRARY

This section contains descriptions of the Form Library functions, in the following sub-sections.

Section	Function description	Binding name
7.1	Process Form	`form_do`
7.2	Reserve Screen For Dialog	`form_dial`
7.3	Draw Alert Box	`form_alert`
7.4	Draw Error Box	`form_error`
7.5	Center Dialog On Screen	`form_center`
7.6	Check Form Keyboard Input	`form_keybd`
7.7	Check Form Button Input	`form_button`

The form library routines are concerned with the use of object trees (as described in the introduction to section 6) to obtain information or selections from the user. Object trees used for this purpose are known as forms, as they act like paper forms which the user has to fill in; they are also known as dialog boxes: particularly simple forms which simply request one of several possible responses to a question.

The majority of forms that an application will use can be easily specified using the object trees described in section 6, and the manner in which the various objects are to interact with the mouse can be completely specified using the `ob_flags` and `ob_state` fields of the object headers. In this case, the user interaction with the form can be done using the function `form_do`, described in section 7.1. This handles such things as radio buttons, selecting and editing text fields, selecting buttons, and returning when an EXIT or TOUCHEXIT object is selected. However, should the application require more sophisticated forms – for example, slider bars that can be dragged to select a value, forms within a window that can be moved, or forms which allow processing to continue in the background while waiting for a user response – then the application will have to assume control of the user's interaction with the form.

Two functions are provided in GEM version 2.0 to assist the processing of mouse clicks or key presses in selecting objects – see `form_keybd` (section 7.6) and `form_button` (section 7.7), but the application will still need to do a fair amount of work. An application which performs its own form handling should conform to the conventions used by `form_do` – for example the use of the up and down arrows or tab and backtab to select the next or previous text field to edit, and the selection of the object whose default bit is set when the enter key is pressed. If `form_keybd` is used much of this will be handled automatically. The treatment of the SELECTABLE, EXIT, TOUCHEXIT, and RBUTTON flags should also be consistent with that described in section 6, as performed by `form_do`.

In order to display and use a form, whether it is to be processed using `form_do` or by the application, the following steps should be used :-

1. The form must be either created using the functions in the object library, or loaded from a resource file, as described in section 12.

2. The coordinates of the form should be calculated and placed in the object header of the root – for a normal form (i.e. not in a window) displayed centrally, this can be done simply by using `form_center` (section 7.5).

3. The portion of the screen which the dialog will occupy should be reserved using `form_dial` (section 7.2). This is not necessary for a form displayed in a window. The screen area can be determined either from the values returned by `form_center` (section 7.5) if used, or by inspecting the coordinates in the root object's header record.

4. The object tree describing the form should be drawn, using `objc_draw` (section 6.3).

5. The form should be processed by calling `form_do` (section 7.1) or by the application's own code, until an exit condition is satisfied.

6. The screen area reserved under step 3 must be released using `form_dial`, causing any screen areas obscured to be redrawn (or redraw messages sent to the owners of any windows affected).

For simple dialogs, an alert or error box may be a more suitable option. The advantage of these is that they do not cause redraw messages to be issued, as the area obscured is saved to a buffer before the alert box is drawn, and restored afterwards. The alert box is therefore removed from the display very much faster than a dialog which obscures windows would be. The price to be paid for this is the lack of flexibility compared to the dialog box. See sections 7.3 and 7.4 for information on how to use alert and error boxes.

7.1 Process Form `form_do`

Process Form is used to process a form or dialog box, taking control of all mouse interaction with buttons and selection and editing of editable text fields. The application is suspended until the user exits from the form by clicking on an exit button, or pressing return if a button is marked as default.

7.1.1 Definition

The Prospero C definition of Process Form is :

```
WORD form_do(OBJECT *form, WORD start);
```

7.1.2 Purpose

This function is used by an application to allow a user to interact with a dialog box – editing text, selecting buttons and radio buttons and so on – and returns when the user clicks on an exit button (or presses Enter if there is an object with its DEFAULT bit set in the `ob_flags` field). All interaction is managed by GEM AES without intervention by the application.

The parameter `start` gives the index of an editable text field which is to be the first one to be edited. The application should pass a zero here if there are no editable text fields.

The function returns the index of the object which caused `form_do` to return – usually an exit button which was clicked on. This object will have its SELECTED state bit set, which will cause it to be drawn highlighted the next time the form is drawn. It is therefore a good idea for the application to reset this bit immediately, using `objc_change` (section 6.8), `objc_newstate` (section 6.10) or directly as shown in the example. Other objects in the tree may have their SELECTED bits set if the user clicked on them while interacting with the form – these will normally be tested by the application to discover what the user requested. Unless the application wants these objects to be pre-selected the next time the object is drawn, it is a good idea to reset the state of all objects in the tree.

The use of `form_do` for interaction with forms is recommended wherever possible, as it greatly simplifies the task. Should the facilities available be inadequate, or should the application wish to continue background processing while awaiting user response, the form must be processed by the application – see the descriptions of the `objc_edit` (section 6.7), `form_keybd` (section 7.6) and `form_button` (section 7.7) for an indication of how this can be done.

7.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`form`	*OBJECT**	**Tree describing form** The pointer to the array containing the tree describing the form which is to be processed.
`start`	*WORD*	**Index of first editable text field** The index within the form of an editable text field which is to be made active when the form is displayed. A value of zero can be used if there are no editable text fields.

7.1.4 Function Result

The value returned is the object index of the object which caused the form to exit. This will be an object (normally a button) with its EXIT or TOUCHEXIT bit set in the `ob_flags` field, which the user selected by clicking on it or (if the object has its DEFAULT bit set) pressing Return.

7.1.5 Example

```
#define NORMAL 0        /* In AESBIND.H file */

#define button1 10      /* Sample constants .. */
#define button2 11      /* .. provided by resource editor */
#define button3 12
#define okbutt  20
#define cancel  21

OBJECT *form;
WORD exit_object, b1state, b23state;

  /* NB the code to reserve screen space and draw the
     form is not shown - see form_dial in section 7.2 */

  /* Interact with a form */
  exit_object = form_do(form, 0);

  /* Reset state of exit button */
  form[exit_object].ob_state = NORMAL;

  if (exit_object == okbutt)
    {  /* User clicked ok - read form */
       b1state = objc_state(form, button1);
       objc_newstate(form, button1, NORMAL);  /* Unselect it */

       /* Buttons 2 and 3 are radio buttons -
          see which one is set */
       b23state = (objc_state(form, button2) != NORMAL);
       /* Don't reset state - leave selected for next time*/
    }
  else                          /* User clicked cancel */
    {                           /* Undo all form changes */
      objc_newstate (form, button1, NORMAL);
      if (b23state)
        { objc_newstate (form, button2, SELECTED);
          objc_newstate (form, button3, NORMAL);
        }
      else
        { objc_newstate (form, button2, NORMAL);
          objc_newstate (form, button3, SELECTED);
        }
    }
```

7.2 Reserve Screen For Dialog `form_dial`

Reserve Screen For Dialog is used to signal that a portion of the screen is about to be used for a dialog box, or to release that portion of the screen. In GEM version 1.1, it can also be used to draw a zoom box expanding or shrinking – these features have been removed from GEM version 2.0.

7.2.1 Definition

The Prospero C definition of Reserve Screen For Dialog is :

```
WORD form_dial(WORD dtype,
               WORD lx, WORD ly, WORD lw, WORD lh,
               WORD x, WORD y, WORD w, WORD h);
```

7.2.2 Purpose

This function is used by an application to reserve or free a portion of the screen for use by a dialog box. In GEM version 1.1 it can also be used to draw expanding or shrinking zoom boxes to make a dialog seem to be appearing from a particular point. These have been removed from GEM version 2.0, but see the functions `xgrf_stepcalc` and `xgrf_2box` in section 14 for how to draw zoom boxes. When `form_dial` is used to free a portion of the screen, it will cause a redraw message to be sent to the application which owns any windows which were covered by the specified rectangle, and the desktop background will be redrawn where it is not covered by windows. This feature can be used to mark an area of the screen as 'dirty' and cause it to be redrawn, even if a dialog has not been produced.

7.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`dtype`	*WORD*	**Action required** The required action the function is to perform, as follows :- 0 (FMD_START) Reserve screen space for a dialog box whose size is given by the parameters `x`, `y`, `w`, and `h`.

1 (FMD_GROW)
Draw box expanding from that described by the parameters `lx`, `ly`, `lw` and `lh` to that described by the parameters `x`, `y`, `w`, and `h`. (Not available in GEM version 2.0)

2 (FMD_SHRINK)
Draw box shrinking from that described by the parameters `x`, `y`, `w`, and `h` to that described by the parameters `lx`, `ly`, `lw` and `lh`. (Not available in GEM version 2.0)

3 (FMD_FINISH)
Free the screen space described by the parameters `x`, `y`, `w`, and `h`, and cause it to be redrawn by the screen manager or the owner of any windows it covers.

The constants named above are defined as macros in the file AESBIND.H.

`lx`	*WORD*	**Small box X coordinate**
`ly`	*WORD*	**Small box Y coordinate**
`lw`	*WORD*	**Small box width**
`lh`	*WORD*	**Small box height**

The coordinates and size of the little box for expanding from and shrinking to (GEM version 1.1). In GEM version 2.0, these parameters are reserved, and values of zero should be passed.

`x`	*WORD*	**Dialog box X coordinate**
`y`	*WORD*	**Dialog box Y coordinate**
`w`	*WORD*	**Dialog box width**
`h`	*WORD*	**Dialog box height**

The coordinates and size of the screen area to be reserved or freed, or (in GEM version 1.1) of the large box to be expanded to or shrunk from.

7.2.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

7.2.5 Example

```
#define NORMAL 0                    /* In AESBIND.H */

OBJECT *form;
WORD exit_object, x, y, w, h;

  form_center(form, &x, &y, &w, &h);
  /* Reserve screen space for dialog */
  form_dial(0, 0, 0, 0, 0, x, y, w, h);

  /* Draw the form */
  objc_draw(form, 0, 32767, x, y, w, h);
  /* Process the form */
  exit_object = form_do(form, 0);
  /* Reset exit button */
  form[exit_object].ob_state = NORMAL;

  /* Release space, to cause redraws */
  form_dial(3, 0, 0, 0, 0, x, y, w, h);

  /* Now proceed to check form etc. */
```

7.3 Draw Alert Box `form_alert`

Draw Alert Box is used to place an alert box on the screen, containing one of three icons, up to five lines of text, and up to 3 exit buttons which the user may select among.

7.3.1 Definition

The Prospero C definition of Draw Alert Box is :

```
WORD form_alert(WORD defbut, const char *astring);
```

7.3.2 Purpose

This function is used by an application to draw an alert box, wait until the user selects a button, then return, restoring the area covered by the box and indicating which button the user selected. For many interactions with the user, an application does not need the more sophisticated features available in a dialog; the alert box is easier to use, faster, and does not cause redraw messages as the area under the alert is saved by GEM before the alert is drawn, and restored afterwards.

An alert box is a very simple dialog with an icon (optional), up to 5 lines of text, and up to three exit buttons, no more than 20 characters in length. One of the buttons may be nominated as the default button using the parameter `defbut`. If there is a default button, pressing Return or Enter will have the same effect as selecting that button with the mouse. The function result indicates which button was selected.

The format of the alert is defined by the the null-terminated string pointed to by the parameter `astring`, which consists of 3 parts, each enclosed by square brackets. The first part contains a digit in the range 0 to 3 indicating which icon is to be used in the alert :-

0 – no icon
1 – NOTE icon
2 – WAIT icon
3 – STOP icon

The second part contains the text to be displayed in the alert. There may be up to 5 lines of text, each of up to 40 characters, separated by vertical bars (|).

The third part contains the names of the buttons – there may be up to 3, each containing up to 20 characters, and again separated by vertical bars.

7.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
defbut	*WORD*	**Default button** The default exit button, which will be selected if Return or Enter is pressed. A value of zero indicates that there is to be no default, and pressing Return or Enter will have no effect. A value in the range 1 to 3 gives the position in the list of buttons of the default button.
astring	*const char **	**Alert definition string** The string containing the definition of the alert, as described above in 7.3.2. Any string which does not conform precisely to the above description is liable to cause GEM to fall over in a heap.

7.3.4 Function Result

The value returned is in the range 1 to 3, and gives the position in the alert's list of buttons of the button which the user selected.

7.3.5 Example

```
form_alert(1,"[1][This is an alert][ OK ]");
/* OK button is default, NOTE icon */

if (form_alert(2, "[0]"
                  "[Do you want | to continue ?]"
                  "[ YES | NO ]")
               == 2)
  exit(0);
/* NO button is default, no icon */
```

7.4 Draw Error Box `form_error`

Draw Error Box is used to place an error box on the screen, which allows operating system errors to be reported in a GEM type manner. The form of the error box is determined by GEM AES depending upon the error number.

7.4.1 Definition

The Prospero C definition of Draw Error Box is :

```
WORD form_error(WORD errnum);
```

7.4.2 Purpose

This function is used by an application to report operating system errors to the user in the form of GEM alert boxes describing the error, and inviting the user to take appropriate action. Some errors will be recognised and a useful description given, others will simply be reported as an error number with an OK box. Some errors may allow the user a choice of responses – the response chosen can be determined from the function result.

7.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
errnum	*WORD*	**Error number** The operating system error code.

7.4.4 Function Result

The value returned will be the number of the exit button which the user selected in the error box. The meaning will vary depending upon the error number.

7.4.5 Example

```
WORD dos_err_code;

  if (dos_err_code)
    form_error(dos_err_code);
  else
    /* no error ... */
```

7.5 Center Dialog On Screen `form_center`

Center Dialog on Screen is used to calculate the rectangle which a dialog will occupy when placed in the center of the screen, and sets the `ob_x` and `ob_y` fields in the dialog's root object header so that the dialog will be displayed in this position.

7.5.1 Definition

The Prospero C definition of Center Dialog on Screen is :

```
WORD form_center(OBJECT *tree,
                 WORD *pcx, WORD *pcy,
                 WORD *pcw, WORD *pch);
```

7.5.2 Purpose

This function is used by an application to calculate the coordinates of a dialog box, and to cause the dialog to be displayed in the center of the screen. It will normally be used to obtain the values to use for `form_dial` (section 7.2) when a dialog box is to be drawn. If the application wishes to display the dialog other than in the center, it must calculate the screen offsets itself, and place them in the `ob_x` and `ob_y` fields of the root object header. In order to do this in a manner that will be portable to all devices of whatever resolution, it is often simpler to use `form_center` to place the form in the center, then adjust the coordinates relative to the center, perhaps just altering the `ob_y` field to display the dialog nearer the top or bottom of the screen.

7.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
tree	*OBJECT* *	**Dialog tree** The tree containing the dialog to be centered.
pcx	*WORD* *	**Dialog X coordinate**
pcy	*WORD* *	**Dialog Y coordinate**
pcw	*WORD* *	**Dialog width**
pch	*WORD* *	**Dialog height** These parameters point to objects into which the function writes the coordinates, width and height of the centered dialog.

7.5.4 Function Result

The value returned is reserved, and is always one.

7.5.5 Example

```
OBJECT *my_form;
WORD x, y, w, h;

  /* Center box */
  form_center(my_form, &x, &y, &w, &h);

  /* Move it up a bit - note that my_form points to
     the root object */
  y /=  2;
  my_form->ob_y = y;

  /* Draw it */
  form_dial(0, 0, 0, 0, 0, x, y, w, h);
  objc_draw(my_form, 0, 32767, x, y, w, h);
```

7.6 Check Form Keyboard Input `form_keybd`

Check Form Keyboard Input is used by applications which are doing their own form processing rather than using `form_do` (see section 7.1). It checks the given input from the keyboard to see whether it is one of tab, backtab, uparrow, downarrow or return, which have special effects on forms, and returns the index of the object selected by the character typed. It is not available in GEM version 1.1.

7.6.1 Definition

The Prospero C definition of Check Form Keyboard Input is :

```
WORD form_keybd(OBJECT *form,
                WORD obj, WORD nextobj, WORD inkey,
                WORD *newobj, WORD *outkey);
```

7.6.2 Purpose

This function is used by an application to check whether the user used one of the special form handling keys tab, backtab, uparrow, downarrow or return, and to discover what the effect of that key should be on the form that the application is processing. It is only used when an application is doing its own form processing rather than using `form_do` (section 7.1), perhaps because it wants some special features in the form, or it wants to continue processing while waiting for input. Its use is closely connected with that of `objc_edit` (section 6.7), as its primary purpose is to see which editable text object the user has selected. This function is not provided in GEM version 1.1, where it is up to the application to decide how to handle these keys. A fairly primitive example of how this could be done is given in 6.7.5.

7.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`form`	*OBJECT* *	**Dialog tree** The tree containing the dialog being processed.
`obj`	*WORD*	**Current editable text object** The object index of the current editable text object. This will be used when calculating which editable text object should become current as a result of the key press.
`nextobj`	*WORD*	**Reserved** Reserved for future use – a value of zero should be passed.
`inkey`	*WORD*	**Key press to be checked** The key stroke which has been returned by `evnt_multi` (section 4.6) or `evnt_keybd` (section 4.1), and which is to be checked for its effect on the form.
`newobj`	*WORD* *	**New current object** This parameter points to the object which returns the object index of the new current object, after the key press has had its effect. This will usually be an editable text field, unless the key was Return or Enter, in which case the object with its DEFAULT bit set will be returned (and will be displayed highlighted on the screen). Note that this function does not cause the new text to be initialized or the old one to be exited – this should be done using `objc_edit`, as in the example.
`outkey`	*WORD* *	**Key press value returned** This parameter points to an object which will return zero if the key press was one of the special form handling keys, otherwise it will return the key code ready to pass to `objc_edit`.

7.6.4 Function Result

The result returned is one if the key tested did not cause an exit condition to be satisfied, or zero if the key was Return or Enter and an object with its DEFAULT bit set was found. The index of the object will be returned in `newobj`.

7.6.5 Example

```
#define text1obj  20
#define text2obj  24
#define text3obj  28
   /* Sample constants provided by resource editor when
      the form was created */

OBJECT *form,
WORD current, new_obj, key, pos;
int not_exit;

  current = text1obj;
  /* Initialize first editable text field */
  objc_edit(form, current, 0, &pos, ED_INIT);

  do
  { key = evnt_keybd();
    not_exit = form_keybd(form, current, 0,
                          key, &new_obj, &key);
    if (key == 0)
      { /* Remove cursor from current text field */
        objc_edit(form, current, key, &pos, ED_END);
        current = new_obj;
        if (not_exit)
          /* Initialize new text field */
          objc_edit(form, current, key, &pos, ED_INIT);
      }
    else
      /* Process the character typed */
      objc_edit(form, current, key, &pos, ED_CHAR);
  } while (not_exit);
```

7.7 Check Form Button Input `form_button`

Check Form Button Input is used by applications which are doing their own form processing rather than using `form_do` (see section 7.1). It waits for a specified number of mouse clicks and uses those mouse clicks to determine which the object selected was, altering the screen display accordingly. It is not available in GEM version 1.1.

7.7.1 Definition

The Prospero C definition of Check Form Button Input is :

```
WORD form_button(OBJECT *form,
                 WORD obj, WORD clicks,
                 WORD *nextobj);
```

7.7.2 Purpose

This function is used by an application to handle mouse interaction with a form. The current object in the form is passed as a parameter, and the number of clicks for which the function is to wait. When the clicks have been received, the function returns, giving the index of the object that was selected by the mouse click (and displaying it as selected) and indicating whether it has the EXIT or TOUCHEXIT bit set in its `ob_flags` field.

According to the GEM documentation, only editable text objects can be selected by this routine, which makes it of very little use, because since it waits for a mouse click before returning, key presses will not be detected.

7.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`form`	*OBJECT **	**Dialog tree** The tree containing the dialog being processed.

`obj` *WORD* **Current editable text object**

The object index of the current editable text object. This will be used when calculating which editable text object should become current as a result of the button click.

`clicks` *WORD* **Number of clicks**

The number of clicks for which the application is waiting.

`nextobj` *WORD ** **Object selected**

Points to a variable which returns the index of the object the user clicked on, or zero if the object is hidden, disabled or not editable.

7.7.4 Function Result

The result returned is zero if the object selected has its EXIT or TOUCHEXIT bit set, otherwise one.

7.7.5 Example

```
OBJECT *form;
WORD current, result;

  /* Get new current object after a click */
  result = form_button(form, current, 1, &current);
```

8 GRAPHICS LIBRARY

This section contains descriptions of the Graphics Library functions, in the following sub-sections.

Section	Function description	Binding name
8.1	Draw Rubberbanded Box	`graf_rubbox`
8.2	Drag Box Within Rectangle	`graf_dragbox`
8.3	Draw Moving Box	`graf_mbox`
8.4	Draw Zoom Boxes	`graf_growbox` `graf_shrinkbox`
8.5	Track Mouse In Box	`graf_watchbox`
8.6	Track Sliding Box	`graf_slidebox`
8.7	Obtain Workstation Handle	`graf_handle`
8.8	Set Mouse Form	`graf_mouse`
8.9	Return Mouse State	`graf_mkstate`

The functions in the graphics library are concerned with miscellaneous graphics drawing and mouse activities. They are based upon GEM VDI functions, which are described further in the VDI manual. Where similar functions are provided in both GEM AES and GEM VDI, applications which use the AES should use the AES functions rather than the VDI functions.

Many of the routines in this section could be used by an application which is handling a user's interaction with a complicated form (e.g. `graf_slidebox`, `graf_watchbox`), or one which allowed icons to be selected and moved around the desktop or window area (e.g. `graf_rubbox`, `graf_dragbox`, `graf_slidebox`). See the relevant sub-section for further details.

8.1 Draw Rubberbanded Box `graf_rubbox`

Draw Rubberbanded Box is used to allow the user to select an area of the screen by 'rubberbanding' a box around it. This means that the box drawn is constantly updated whenever the mouse is moved so that the mouse position is at the bottom right corner (the top left corner is fixed). In GEM version 1.1 this function was called `graf_rubberbox`, but is otherwise unchanged. The Prospero C function is called `graf_rubbox` for both versions to aid portability.

8.1.1 Definition

The Prospero C definition of Draw Rubberbanded Box is :

```
WORD graf_rubbox(WORD xorigin, WORD yorigin,
                 WORD wmin, WORD hmin,
                 WORD *pwend, WORD *phend);
```

8.1.2 Purpose

This function is used by an application to allow the user to select an area of the screen by dragging a rubberbanded box around it. The application should call this function when it detects the leftmost mouse button being pressed, passing the coordinates of the mouse at the time. The function will track the mouse position and redraw the rectangle as necessary until the mouse button is released, and return the width and height of the box the user selected. The minimum width and height that the box may take are specified by the application in the parameters `wmin` and `hmin`.

8.1.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`xorigin`	*WORD*	**X coordinate of box**
`yorigin`	*WORD*	**Y coordinate of box**
		The x and y coordinates of the top left hand corner of the box, which remains fixed throughout the rubberbanding operation.
`wmin`	*WORD*	**Rubber box minimum width**
`hmin`	*WORD*	**Rubber box minimum height**
		The minimum width and height which the rubberbanded box is allowed to take on.

pwend	*WORD* *	**Rubber box final width**
phend	*WORD* *	**Rubber box final height**

The final width and height of the rectangle when the mouse button was released is returned in the objects pointed to by these parameters. The values will be greater than or equal to the minimum width and height passed in `wmin` and `hmin`.

8.1.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

8.1.5 Example

```
WORD x, y, w, h, dummy;
OBJECT *the_form;

  /* Wait for left hand button down */
  evnt_button(1, 1, 1, &x, &y, &dummy, &dummy);
  if (objc_find(the_form, 0, 2, x, y) != -1)
    { /* User selected an object */
      .
      .
    }
  else
    { /* Clicked in space - select area */
      graf_rubbox(x, y, 10, 10, &w, &h);
      /* x, y, w, h give rectangle user selected */
      .
      .
    }
```

8.2 Drag Box Within Rectangle `graf_dragbox`

Drag Box Within Rectangle is used to allow the user to select the position of a box on the screen, within constraints defined by the application. See also `graf_slidebox` in section 8.6.

8.2.1 Definition

The Prospero C definition of Drag Box Within Rectangle is :

```
WORD graf_dragbox(WORD w, WORD h, WORD sx, WORD sy,
                  WORD xc, WORD yc, WORD wc, WORD hc,
                  WORD *pdx, WORD *pdy);
```

8.2.2 Purpose

This function is used by an application to allow the user to position a box anywhere within a specified area of the screen. The box is drawn with a dotted line as it is moved, so that the user gets visual feedback of the position being selected. The function first waits until the leftmost mouse button is depressed (this may already be the case). The offset of the mouse relative to the box to be dragged is then calculated, and as the mouse is moved, the box is drawn in the position required to keep this offset the same (though it will never move outside the bounding rectangle). When the mouse button is released, the final coordinates of the box are returned in the objects pointed to by `pdx` and `pdy`.

8.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
w	*WORD*	**Width of box to be dragged**
h	*WORD*	**Height of box to be dragged**
		The width and height of the box which the user is to drag.
sx	*WORD*	**Initial X coordinate of box**
sy	*WORD*	**Initial Y coordinate of box**
		The initial x and y coordinates of the box which the user is to drag.
xc	*WORD*	**Bounding box X coordinate**
yc	*WORD*	**Bounding box Y coordinate**
wc	*WORD*	**Bounding box width**
hc	*WORD*	**Bounding box height**
		The coordinates, width and height of the bounding box within which the box being dragged is constrained.
pdx	*WORD* *	**Final X coordinate of box**
pdy	*WORD* *	**Final Y coordinate of box**
		Used to return the x and y coordinates of the box when the user released the mouse button.

8.2.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

8.2.5 Example

```
/* A simple example of how an application might allow
   the user to move objects around the screen. The code
   to remove and redraw the object is not included */

WORD x, y, w, h, x1, y1, dummy;
OBJECT *the_form;

  /* Wait for left hand button down */
  evnt_button(1, 1, 1, &x, &y, &dummy, &dummy);

  obj = objc_find(the_form, 0, 1, x, y);
  if (obj > 0)
    { /* User selected a child of the root */
      objc_offset(the_form, obj, &x, &y);
      w = the_form[obj].ob_w;
      h = the_form[obj].ob_h;
      /* x, y, w, h are object area*/

     /* Get new position within root area */

     graf_dragbox(w, h, x, y,
                  the_form->ob_x, the_form->ob_y,
                  the_form->ob_w, the_form->ob_h,
                  &x1, &y1);
     x = x1 - the_form->ob_x;
     y = y1 - the_form->ob_y;

     /* Now set new object position */
     the_form[obj]->ob_x = x;
     the_form[obj]->ob_y = y;
    }
```

8.3 Draw Moving Box `graf_mbox`

Draw Moving Box is used to draw the outline of a box moving from one position to another, without changing size. This function was known as `graf_movebox` in GEM version 1.1, but is otherwise unchanged. The Prospero C function is called `graf_mbox` for both versions, to aid portability.

8.3.1 Definition

The Prospero C definition of Draw Moving Box is :

```
WORD graf_mbox(WORD w, WORD h, WORD srcx, WORD srcy,
               WORD dstx, WORD dsty);
```

8.3.2 Purpose

This function is used by an application to draw an image of a box moving from one screen position to another. When the function has finished, the screen display will be unchanged. This might be used to give a little animation when a user requests that an object be moved from one point to another, and provide a little more visual indication of what has happened than simply removing the object from one position and redrawing it in another.

8.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
w	*WORD*	**Width of moving box**
h	*WORD*	**Height of moving box**
		The width and height of the moving box.
srcx	*WORD*	**Initial X coordinate of box**
srcy	*WORD*	**Initial Y coordinate of box**
		The initial x and y coordinates of the moving box.
dstx	*WORD*	**Final X coordinate of box**
dsty	*WORD*	**Final Y coordinate of box**
		The final x and y coordinates of the moving box.

8.3.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

8.3.5 Example

```
WORD x1, y1, x2, y2;

/* Set up start and end positions in x1, y1, x2, y2 */
   .
   .

/* Draw the moving box */
  graf_mbox(20, 20, x1, y1, x2, y2);
```

8.4 Draw Zoom Boxes

`graf_growbox`
`graf_shrinkbox`

These functions are used to draw the outlines of a sequence of boxes expanding or shrinking from one rectangle to another. These might be used to give a user visual feedback about, for example, which object's selection had caused a dialog or window to be opened. However, such effects are time consuming, and can be irritating to users, so a friendly application might provide a way of disabling them. All such animation effects have been removed from the GEM Desktop in version 2.0, in the interests of both speed and space, and these functions are not supported by GEM version 2.0 (though see `xgrf_stepcalc` and `xgrf_2box` in section 14).

8.4.1 Definition

The Prospero C definitions of Draw Zoom Boxes are :

```
WORD graf_growbox(WORD sx, WORD sy, WORD sw, WORD sh,
                  WORD fx, WORD fy, WORD fw, WORD fh);
WORD graf_shrinkbox(WORD fx, WORD fy, WORD fw,WORD fh,
                    WORD sx, WORD sy, WORD sw,WORD sh);
```

8.4.2 Purpose

These functions are used by an application to draw an image of a box expanding or shrinking from one rectangle to another, so that dialogs or windows can seem to appear from a particular point. Use `graf_growbox` if the final box is larger than the initial box, otherwise use `graf_shrinkbox`.

These functions are not provided in GEM version 2.0.

8.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
sx	*WORD*	**X coordinate of starting box**
sy	*WORD*	**Y coordinate of starting box**
sw	*WORD*	**Width of starting box**
sh	*WORD*	**Height of starting box**
		The initial x and y coordinates and size of the expanding or shrinking boxes.
fx	*WORD*	**X coordinate of finishing box**
fy	*WORD*	**Y coordinate of finishing box**
fw	*WORD*	**Width of finishing box**
fh	*WORD*	**Height of finishing box**
		The final x and y coordinates and size of the expanding or shrinking boxes.

8.4.4 Function Result

The value returned will be zero if an error occurred, or greater than zero if no error was detected.

8.4.5 Example

```
WORD ox, oy, wx, wy, ww, wh;
OBJECT *the_tree;
WORD obj_selected;
WORD my_window;
WORD animating;

  objc_offset(the_tree, obj_selected, &ox, &oy);
  if (animating)
    /* Make window appear to grow out of object */
    graf_growbox(ox, oy, 10, 10, wx, wy, ww, wh);
  wind_open(my_window, wx, wy, ww, wh);
  . .
  wind_close(my_window);
  if (animating)
    /* Make window appear to shrink back into object */
    graf_shrinkbox(ox, oy, 10, 10, wx, wy, ww, wh);
```

8.5 Track Mouse In Box `graf_watchbox`

Track Mouse In Box is used to track the mouse as it moves in and out of an object box, and alter the object's `ob_state` field (and therefore its appearance) accordingly. This is used for example when processing a form – if the user clicks in a box, the mouse can be tracked in and out of the button until the mouse button is released, to give the user a chance to cancel the selection by moving out of the box before releasing the button.

8.5.1 Definition

The Prospero C definition of Track Mouse In Box is :

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

8.5.2 Purpose

This function is used by an application to change the state and appearance of an object (usually a box) according to whether the mouse is inside or outside the box. The application should call this when the mouse button is down, usually inside the box in question. The function will set the state and appearance of the box to either `instate` or `outstate` depending on whether the mouse is inside or outside the box, and return when the mouse button is released, indicating whether the button was inside or outside the box at the time. See section 6 for details of the `ob_state` field, and suitable values for `instate` and `outstate`.

8.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT* *	**Tree containing object** The tree containing the object in question.
`obj`	*WORD*	**Object being tracked** The index within the tree of the object being tracked.
`instate`	*WORD*	**State when mouse in box** The value of the object's `ob_state` field when the mouse is inside the box. Typically this will be the same as the original state of the object when the mouse click was detected, except with the selected bit set or toggled, depending on what sort of object it is.
`outstate`	*WORD*	**State when mouse out of box** The value of the object's `ob_state` field when the mouse is outside the box. Typically this will be the original state of the object when the mouse click was detected.

8.5.4 Function Result

The result is one if the mouse was inside the box when the button was released, otherwise zero.

8.5.5 Example

```
OBJECT *form;
WORD mx, my, obj, dummy;

  while (1)
   { /* Wait for 1 click of left hand button */
     evnt_button(1, 1, 1, &mx, &my, &dummy, &dummy);

     /* Search form to see where user clicked */
     obj = objc_find(form, 0, 5, mx, my);

    if (obj == -1)
      { /* outside form altogether - beep */
        ...
      }
    else
      { WORD flags = objc_flags(form, obj),
             state = objc_state(form, obj);

        if (flags & TOUCHEXIT)
          /* Exit immediately if TOUCHEXIT set */
          break;

        /* If it's selectable, track until button up */
        if (flags & SELECTABLE)
          if (graf_watchbox(form, obj,
                            state ^ SELECTED,
                            state))
            /* Button released within object */
            if (flags & EXIT)
              break;
      }

   }
```

8.6 Track Sliding Box `graf_slidebox`

Track Sliding Box is used to track the position of one box inside another box as the mouse is moved, in the same way as the slider bars of a window work.

8.6.1 Definition

The Prospero C definition of Track Sliding Box is :

```
WORD graf_slidebox(OBJECT *tree,
                   WORD parent, WORD obj, WORD isvert);
```

8.6.2 Purpose

This function is used by an application to allow a user to set the position of a box within another box either vertically or horizontally, by moving the mouse and releasing the button when the sliding box is in the required position. The application should call this function when the mouse button is down; GEM AES will note the position of the mouse relative to the sliding box, and as the mouse is moved the box will move within its constraining box to maintain this offset as far as possible. When the mouse button is released, the function returns the relative position of the sliding box within its constraining box. Both the sliding box and its constraining box are objects in the same tree, the constraining box being the parent of the sliding box.

The slider bars of GEM windows provide a good example of how such sliding boxes operate, though they can also be used in forms (if the application is prepared to process the forms itself) or within windows etc.

8.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT**	**Tree containing box objects** The tree containing the parent and sliding boxes.
`parent`	*WORD*	**Index of parent box** The index within the tree of the parent box, which defines the range of movement of the sliding box.

`obj` *WORD* **Index of sliding box**

The index within the tree of the sliding box, which should be a child of the parent box.

`isvert` *WORD* **Vertical flag**

A flag indicating whether the movement is to be vertical (`isvert` = 1) or horizontal (`isvert` = 0).

8.6.4 Function Result

The result returned is a WORD in the range 0 to 1000 giving the relative position of the slider within the parent – 0 means at the top or left, while 1000 means at the bottom or right.

8.6.5 Example

```
#define slider 20
#define parent 19
    /* Indices of slider and parent in form,
       perhaps provided by resource editor */

OBJECT *form;
WORD mx, my, dummy, obj_selected, val;

/* Wait for 1 click of left hand button, ignore keys */
evnt_button(1, 1, 1, &mx, &my, &dummy, &dummy);

/* Search form to see where clicked */
obj_selected = objc_find(form, 0, 5, mx, my);

if (obj_selected == slider)
  /* Clicked on slider, so slide it */
  val = graf_slidebox(form, parent, slider, 1);
```

8.7 Obtain Workstation Handle `graf_handle`

Obtain Workstation Handle is used to discover the GEM VDI handle of the currently open screen workstation, to which GEM AES output is directed. The application can use this as the handle for its own GEM VDI output to the screen, or use it to open a virtual screen workstation for its own screen output. The latter is preferable, as any attributes set by the application on the virtual workstation will not affect GEM AES's output.

8.7.1 Definition

The Prospero C definition of Obtain Workstation Handle is :

```
WORD graf_handle(WORD *pwchar, WORD *phchar,
                 WORD *pwbox, WORD *phbox);
```

8.7.2 Purpose

This function is used by an application to discover the GEM VDI handle of the screen workstation, so that it can open a virtual workstation and/or make GEM VDI output calls to the screen (see the VDI Bindings manual for details). The sizes of a character cell in the system font used in menus and dialogs, and of a square box large enough to hold such a character, are also returned.

8.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
pwchar	*WORD **	**Character cell width**
phchar	*WORD **	**Character cell height**
		Pointers to objects which return the width and height (in pixels) of a character cell in the system font.
pwbox	*WORD **	**Character box width**
phbox	*WORD **	**Character box height**
		Pointers to objects which return the width and height (in pixels) of a square box large enough to hold a system font character.

8.7.4 Function Result

The result returned is the GEM VDI handle of the currently open screen workstation.

8.7.5 Example

```
WORD handle, dummy, i;
WORD work_in[11], work_out[57];

main()
{ appl_init();
  handle = graf_handle(&dummy, &dummy, &dummy, &dummy);
  for (i = 0; i < 10; i++)
    work_in[i] = 1;      /* Select initial attributes */
  work_in[10] = 2;                    /* Raster coords */
  v_opnvwk(work_in, &handle, work_out);
                /* Open virtual screen workstation */
  .
  .
  v_clsvwk(handle);    /* must close before returning */
  appl_exit();
}
```

8.8 Set Mouse Form `graf_mouse`

Set Mouse Form is used to set the mouse form to one of a set of predefined forms, or to a user defined form, or to hide or show the mouse form.

8.8.1 Definition

The Prospero C definition of Set Mouse Form is :

```
WORD graf_mouse(WORD m_number, WORD m_addr[]);
```

8.8.2 Purpose

This function is used by an application to select the mouse form, to set it to a user defined form, or to control whether the mouse is displayed or hidden. The predefined cursor forms are as follows :-

0 – arrow
1 – text cursor (I bar)
2 – busy cursor (hourglass or bee)
3 – hand with pointing finger
4 – flat hand with extended fingers
5 – thin cross hair
6 – thick cross hair
7 – outlined cross hair

Any cursor form other than the arrow or the busy cursor should only be used within the active window's work area. If an application uses one of these, it must make an `event_multi` (section 4.6) or `evnt_mouse` (section 4.3) call to detect whenever the mouse enters or leaves the active window, and set the cursor form appropriately.

8.8.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
m_number	*WORD*	**Mouse form required** This parameter selects the mouse form required as follows : 0 to 7 – predefined mouse form as defined above 255 – user defined mouse form, described by structure pointed to by the parameter m_addr 256 – hide mouse form 257 – show mouse form Note that two consecutive calls to hide the mouse will require two calls to show it before the mouse reappears.
m_addr	*WORD[]*	**Mouse form address** If the value of the parameter m_number is 255, this parameter should point to an array describing the required mouse form, as described in section 7.6 of the VDI manual. If the value of the parameter m_number is not 255, the value of this parameter is ignored, and NULL may be passed.

8.8.4 Function Result

The value returned is zero if an error occurred, or greater than zero if no error was detected.

8.8.5 Example

```
WORD x, y, w, h;          /* Work area */
WORD mx, my;              /* Mouse coordinates */
WORD dummy;
WORD in_window;           /* a flag */
WORD my_mouse_form[37];

/* A simple example of how to ensure that the user
   mouse form is only used when the mouse lies within
   a window's work area, given by the variables
   x, y, w and h.

   This example uses evnt_mouse for simplicity - to do
   anything useful as well as keeping the mouse form
   correct, evnt_multi should be used - see section
   4.6 */

  /* Set up mouse form first - see VDI manual */

  in_window = 0;
  while (1)
  { /* Wait for mouse to enter or leave window */
    evnt_mouse(in_window, x, y, w, h,
               &mx, &my, &dummy, &dummy);
    in_window = !in_window;
    if (in_window)
      /* Own mouse inside */
      graf_mouse(255, my_mouse_form)
    else
      /* Arrow mouse outside */
      graf_mouse(0, NULL);
  }
```

8.9 Return Mouse State `graf_mkstate`

Return Mouse State is used to discover the current mouse location, the current state of the mouse buttons, and the state of the control, shift and alt keys which may be used to modify the interpretation of mouse behavior.

8.9.1 Definition

The Prospero C definition of Return Mouse State is :

```
void graf_mkstate(WORD *pmx, ,WORD *pmy,
                  WORD *pmstate, WORD *pkstate);
```

8.9.2 Purpose

This function is used by an application to discover the current position of the mouse, and the current state of its button and the shift, control and alt keys on the keyboard.

8.9.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
pmx	*WORD **	**Mouse X coordinate**
pmy	*WORD **	**Mouse Y coordinate**
		These parameters point to objects which return the x and y coordinates of the current mouse position.
pmstate	*WORD **	**Mouse button state**
		This parameter points to an object which returns the state of the mouse buttons, where each bit represents the state of the corresponding button. A bit value of 1 means the button is down, while 0 means it is up. The least significant bit corresponds to the button on the left.

`pkstate` *WORD ** **Keyboard state**

This parameter points to an object which returns the state of the shift, control and alt keys, where each bit represents a key state (1=key down, 0 = key up) as follows :-

bit 0 – right shift	mask 0x0001
bit 1 – left shift key	mask 0x0002
bit 2 – ctrl key	mask 0x0004
bit 3 – alt key	mask 0x0008

8.9.4 Function Result

There is no function result.

8.9.5 Example

```
WORD mx, my, buttons, keys;

  graf_mouse(&mx, &my, &buttons, &keys);
  if (keys & 4)
    { /* Control key is down */
    }
  else
    { /* Control key is up */
    }
```

9 SCRAP LIBRARY

This section contains descriptions of the Scrap Library functions, in the following sub-sections.

Section	Function description	Binding name
9.1	Read Scrap Directory	`scrp_read`
9.2	Write Scrap Directory	`scrp_write`
9.3	Clear Scrap Directory	`scrp_clear`

These functions are concerned with the management of the desk scrap – this is a mechanism for transferring data between applications by means of a set of files on a disk. Applications should write data to a file called 'SCRAP.*' where the extension indicates the form of the data in the file, in response to a user's cut or copy request. The data in these files may then be read by other applications when they are requested to perform a paste operation. In order to give greater flexibility, the application can write the data to a number of scrap files, in different forms – all scrap files should contain the same information, but the form depends upon the file's extension as follows:

SCRAP.TXT – contains ASCII text strings
SCRAP.CSV – contains comma-separated values
SCRAP.DIF – contains spreadsheet data
SCRAP.GEM – a metafile containing GEM VDI images
SCRAP.IMG – a GEM VDI bit image
SCRAP.DCA – contains data in IBM Document Contents Architecture form
SCRAP.USR – contains data in OEM defined form

When the user performs a cut or copy operation, all scrap files should be deleted from the current scrap directory (either using `scrp_clear` or via Prospero C's `remove` function), then the data being cut or copied is written to the relevant scrap file or files. When a paste operation is requested, the application should check the current scrap directory for a scrap file with a suitable extension, and if one exists, copy the data from that to its own workspace.

It is the responsibility of the application to read, write and create the scrap files. The functions in the scrap library allow an application to determine which directory is being used to hold the scrap files, or to nominate a different directory to be used. By using these functions, all applications will use the scrap in a consistent manner.

9.1 Read Scrap Directory `scrp_read`

Read Scrap Directory is used to discover the current directory for the storage of files containing desk scrap information, and (in GEM version 2.0 only) to discover what scrap files currently exist in the scrap directory.

9.1.1 Definition

The Prospero C definition of Read Scrap Directory is :

```
WORD scrp_read(char pscrap[]);
```

9.1.2 Purpose

This function is used by an application to discover which directory is currently being used to hold the desk scrap files. When performing a paste operation, the application should check the scrap directory to see whether any of the scrap formats it supports are available, by looking in it for a file called SCRAP.???, where the extension takes one of the forms described in the introduction to section 9. For a cut or copy operation, the application would clear all scrap files from the scrap directory, before writing its own scrap file or files. In GEM version 2.0, the task of looking for an appropriate scrap file has been simplified, as a bit map indicating which scrap files are present is returned – see 9.1.4 for details.

9.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
pscrap	*char[]*	**Current scrap directory** This parameter returns the directory specification of the directory currently designated as the scrap directory, in which the application should look for scrap files for a paste operation, or place them for a cut or copy operation.

9.1.4 Function Result

The value returned by GEM AES is of little use in GEM version 1.1 – the standard zero meaning error, positive meaning no error. In GEM version 2.0, a bit map is returned indicating which scrap files are present in the directory as follows:

bit 0 – SCRAP.CSV	(comma-separated values)
bit 1 – SCRAP.TXT	(ASCII text)
bit 2 – SCRAP.GEM	(Metafile VDI images)
bit 3 – SCRAP.IMG	(VDI bit images)
bit 4 – SCRAP.DCA	(IBM Document Content Architecture)
bit 5 – SCRAP.USR	(OEM defined data)

Other bits may be used in subsequent releases of GEM.

A value of –1 may be returned to indicate that no scrap directory has been established. A value of zero indicates that a scrap directory has been nominated, but it contains none of the above scrap files.

9.1.5 Example

```
char scrap_path[80];
WORD files;
FILE *tscrap;

  /* GEM 1.1 version */
  /* User has requested a 'paste' operation */
  scrp_read(scrap_path);
  if (scrap_path[0] != '\0')
  { strcat (scrap_path, "SCRAP.TXT");
    if (access(scrap_path, 0) )  /* SCRAP.TXT exists */
      tscrap = fopen ( scrap_path, "w+");
      /* read data from the file tscrap */

  /* GEM 2.0 version */
  /* User has requested a 'paste' operation */
  files = scrp_read(scrap_path);
  if (files > 0)    /* There is some scrap */
    if (files & 2)  /* SCRAP.TXT exists */
    { strcat(scrap_path, "SCRAP.TXT");
      tscrap = fopen(scrap_path,"w+");
      /* read data from the file tscrap */
```

9.2 Write Scrap Directory `scrp_write`

Write Scrap Directory is used to change the current directory for the storage of files containing desk scrap information.

9.2.1 Definition

The Prospero C definition of Write Scrap Directory is :

```
WORD scrp_write(const char *pscrap);
```

9.2.2 Purpose

This function is used by an application to change the directory currently being used to hold the desk scrap files. As it is entirely up to the application to handle scrap file output and so on, the only effect of this function is to change the value that will be returned by `scrp_read` (see section 9.1). So long as all applications are well behaved and use the `scrp_read` function before looking for scrap files, this function will change where they look. This function should only be used prior to a cut or copy operation, as it will cause the current contents of the scrap to become inaccessible.

This function does not create the directory, but checks that it exists, returning an error code if the specified directory cannot be found.

9.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`pscrap`	*const char **	**New scrap directory** The directory specification of the directory to be designated as the scrap directory, in which all applications should look for scrap files for a paste operation, or place them for a cut or copy operation.

9.2.4 Function Result

A value of zero will be returned to indicate an error, such as the directory does not exist, while a value greater than zero means no error was detected.

9.2.5 Example

```
char scrap_path[80];

  scrp_read(scrap_path);
  if (scrap_path[0] == 'A')          /* On a slow disk */
    { strcpy(scrap_path, "C:\");
      scrap_write(scrap_path);       /* .. so move it */
    }
```

9.3 Clear Scrap Directory `scrp_clear`

Clear Scrap Directory is used to delete all SCRAP.* files from the currently nominated scrap directory. This function is not provided in GEM version 1.1.

9.3.1 Definition

The Prospero C definition of Clear Scrap Directory is :

```
WORD scrp_clear(void);
```

9.3.2 Purpose

This function is used by an application to delete all scrap files from the current scrap directory. There must be a current scrap directory, nominated by this or a previous application using `scrp_write` (see section 9.2). An application might use this when the user has requested a cut or copy operation, prior to writing the new scrap file(s) containing the information being cut or copied.

This function is not provided in GEM version 1.1.

9.3.3 Parameters

There are no parameters.

9.3.4 Function Result

A value of zero is returned to indicate an error, such as there is no scrap directory, while a value greater than zero means no error was detected.

9.3.5 Example

```
char scrap_path[80];
FILE *scrap;

  if (scrp_read(scrap_path))    /* No scrap directory */
    { strcpy (scrap_path, "C:\");
      scrp_write(scrap_path);  /* .. so nominate one */
    }
  scrp_clear();              /* Delete any SCRAP files */
  strcat(scrap_path, "SCRAP.TXT");
  scrap = fopen (scrap_path, "w+");
  /* write the data that the user has cut or copied to
    the stream scrap */
```

10 FILE SELECTOR LIBRARY

This section contains a description of the File Selector function.

Section	Function description	Binding name
10.1	Select File and Directory	`fsel_input`

This function is provided to allow all applications to prompt the user for a filename in a consistent manner. The dialog produced is suitable for selecting both existing and new filenames, and allows the user to see what files exist, and either select one of them or select a new filename by typing it in. The file display can be restricted to those files with a particular extension or extensions.

10.1 Select File and Directory `fsel_input`

Select File and Directory is used to allow a user to select a file and directory using the standard GEM file selector form.

10.1.1 Definition

The Prospero C definition of Select File and Directory is :

```
WORD fsel_input(char pipath[], char pisel[],
                WORD *wasok);
```

10.1.2 Purpose

This function is used by an application to prompt the user for a filename and directory using the standard file selector form. By using this function, all GEM applications provide a uniform interface to users, thus making all GEM applications easier to learn. The standard file selector allows the user to see what files currently exist in each directory, and to change directories and disks. The user can choose one of the existing files listed, or type in a new file name – the list of existing files is still useful so the user can be sure that the selected filename does not already exist.

10.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
pipath	*char []*	**Path specification** Points to a null terminated string containing the path specification and file type specification for file selection. The initial, suggested value placed here by the application before calling the function may be modified by the user when interacting with the form, and the setting when the user exited is returned in the same array. The application should take care to ensure that the pathname is in valid DOS form, as GEM is not very robust and liable to behave unpredictably if given an illegal path. The path should finish with a wildcard file specification indicating which files are to be displayed as available for selection. In GEM 2.0, multiple file selectors of the form '*.TXT,*.DOC' are supported.
pisel	*char []*	**File specification** The file specification selected. The application should place an initial suggested choice in the array pointed to by this parameter before calling the function – this can be an empty string, or perhaps the name of the last file selected. The name of the file selected by the user is returned in the same array when the function returns.
wasok	*WORD **	**OK flag** This parameter points to an object which indicates which exit button was selected. If it returns one, the user exited by clicking OK or pressing Return/Enter, indicating that the choice of filename and directory is to be used. If this parameter returns zero, the user exited by clicking on Cancel, and the application should abort the file operation for which it was obtaining the filename.

10.1.4 Function Result

This function returns a value of zero to indicate an error, while a value greater than zero means no error was detected. The most likely error is insufficient memory, as a two kilobyte buffer is required for temporary storage of the directory while sorting it.

10.1.5 Example

```
#include <stdio.h>
#include <string.h>
#include <aesbind.h>

char current_path[81], temp_path[81], filename[81];
char current_file[13], temp_file[13];
WORD ok;
int  i;
FILE *stream;

  strcpy(temp_path, current_path);
  strcat(temp_path, "*.DOC");
  strcpy(temp_file, current_file);
  fsel_input(temp_path, temp_file, &ok);
  if (ok)
    { /* Values in temp_path and temp_file are valid */
      strcpy(current_file, temp_file);
      i = strlen(temp_path);
      while ((i > 0) && (temp_path[i-1] != '\\'))
        i--;
      if (i > 0)
      { /* Path reasonable */
        /* remove '*.DOC' from path*/
        strcpy(current_path, temp_path);
        current_path[i] = '\0';
        strcpy(filename, current_path);
        strcat(filename, current_file);
        stream = fopen( filename, "w+");
        /* Use the file */
      }
    }
  /* Leave current_path, current_file unchanged if
    CANCEL clicked */
```

11 WINDOW LIBRARY

This section contains descriptions of the Window Library functions, in the following sub-sections.

Section	Function description	Binding name
11.1	Create Window	`wind_create`
11.2	Open Window	`wind_open`
11.3	Close Window	`wind_close`
11.4	Delete Window	`wind_delete`
11.5	Inquire Window Attributes	`wind_get`
11.6	Set Window Attributes	`wind_set`
11.7	Find Window Under Point	`wind_find`
11.8	Start / End Window Update	`wind_update`
11.9	Calculate Window Coordinates	`wind_calc`
11.10	Set Window Title or Info	`wind_title` `wind_info`
11.11	Set New Desktop	`wind_newdesk`

The functions in the Window Library are concerned with creating, displaying and updating windows on the screen. Overlapping windows are a very important feature of the GEM interface, and provide a very powerful way of organising screen output in such a way that different information, perhaps even output by different programs, is clearly distinguishable and there is no confusion about what output comes from where. GEM AES supports up to 8 windows at any one time – however if an application were to use 8 windows, there would be none available for desk accessories, which would therefore not be able to run. It is therefore advisable for an application to restrict itself to perhaps 4 windows, leaving 4 available for use by desk accessories.

Note that the functions `wind_title`, `wind_info` and `wind_newdesk` do not form part of the original Digital Research bindings, but are provided in the Prospero C bindings.

Each window is referred to by a window handle, a number in the range 0 to 8 Window handle 0 is special, and refers to the desktop surface, on top of which all application windows lie. A window's handle is returned by the function `wind_create` (section 11.1) which must be used before the window can be used for any of the other window library calls. The call of `wind_create` does not cause the window to be opened (i.e. displayed on the desktop) – this may be done at any subsequent time using `wind_open` (section 11.2). Once created, a window may be opened and closed any number of times, the window handle and associated AES workspace remaining valid until the application indicates that the window handle is no longer required using the function `wind_delete` (section 11.4). Often an application will always open a window immediately after creating it, and always delete it immediately after closing it. However if, for example, the window was being closed temporarily, to be reopened later, it would not be sensible to release the window handle for reuse when the window was closed, as this might result in there being no handle available when the application came to reopen it.

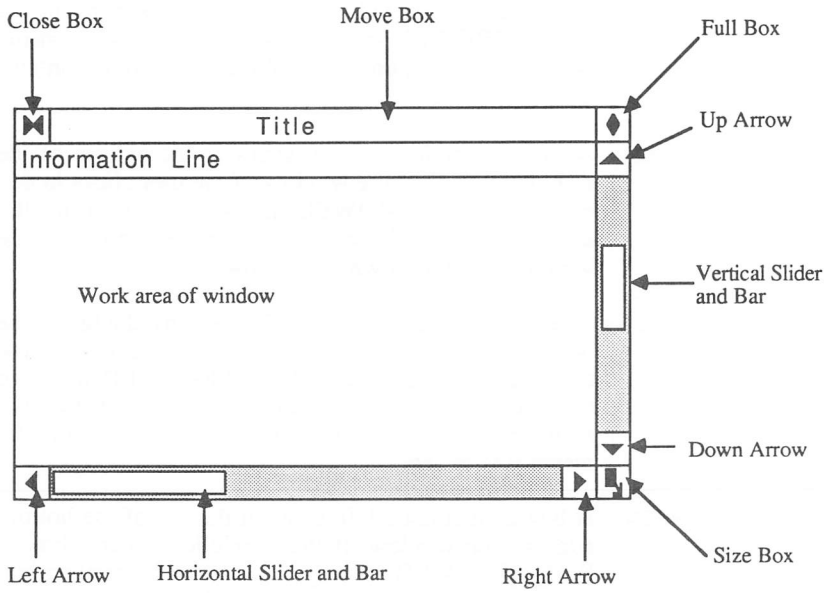
Applications should take care that they correctly handle the case where no windows are available, perhaps by prompting the user to close some windows and try again. An application which does not correctly close and delete its windows before terminating will be a menace to other applications which run after it – if windows are closed but not deleted, there will be less than 8 handles available for subsequent applications, while if they are deleted but not closed, the image will remain on the screen, with control points active, but nowhere to send the messages to when, for example, redraws are required – the result can be very messy! Worst of all are applications which terminate without indicating that a window update is complete – this will disable the menu bar, move and size boxes, and window redraws of the next application, and will normally require the machine to be reset.

A window consists of two independent areas: the work area, which is maintained by the application, and the border area, which is managed by the GEM AES screen manager. The border area consists of a number of separate control areas, each of which causes a different message to be sent to the application when clicked upon, and interacts with the user's mouse behavior in a specific manner. The control areas present on a particular window are specified by the value of the `kind` parameter when the window is created (see section 11.1); each bit of this parameter indicates the presence or absence of a particular window component, and the following constants are provided in the file AESBIND.H to be combined (using `|`) as required:

Value	Name	Component Function
0x0001	NAME	A title bar one character high at the top of the window. The text to be displayed here must be set up using `wind_set` (section 11.6) or `wind_title` (section 11.10) before opening the window. This area does not interact with the mouse, though if the `MOVE` component is present, the move box will coincide with the title bar.
0x0002	CLOSE	A close box in the top left hand corner of the window. If the user clicks in this box, the AES screen manger sends a WM_CLOSED message to the application which owns the window. The application should respond by closing the window, if appropriate.
0x0004	FULL	A full box in the top right hand corner of the window. If the user clicks in this box, the AES screen manger sends a WM_FULLED message to the application which owns the window. The application should respond by expanding the window to its full size or shrinking it back to its previous size.
0x0008	MOVE	A move bar at the top of the window. This bar will also contain the title if the `NAME` component is specified. The user can click in this box, and drag the outline of the window to a new position. When the button is released, the AES screen manger sends a WM_MOVED message to the application which owns the window. The application should respond by moving the window to the requested position if appropriate.
0x0010	INFO	An information line one character high at the top of the window. The text to be displayed here must be set up using `wind_set` (section 11.6) or `wind_info` (section 11.10) before opening the window. This area does not interact with the mouse.
0x0020	SIZE	A size box in the bottom left hand corner of the window. The user can click in this box, and drag the outline of the window to a new size. When the button is released, the AES screen manger sends a WM_SIZED message to the application which owns the window. The application should respond by resizing the window to the requested size if appropriate.

Value	Name	Component Function
0x0040	UPARROW	A box containing an up arrow at the top of the right hand edge of the window. If the user clicks in this box, a WM_ARROWED message is sent to the application, which should respond by scrolling the window contents up by one row.
0x0080	DNARROW	A box containing a down arrow at the bottom of the right hand edge of the window. If the user clicks in this box, a WM_ARROWED message is sent to the application, which should respond by scrolling the window contents down by one row.
0x0100	VSLIDE	A scroll bar and slider along the right hand edge of the window. The user can click in the bar above or below the slider box, causing a WM_ARROWED message requesting a scroll of one page to be sent, or drag the slider box to a new position, causing a WM_VSLID message to be sent.
0x0200	LFARROW	A box containing a left arrow at the left of the bottom edge of the window. If the user clicks in this box, a WM_ARROWED message is sent to the application, which should respond by scrolling the window contents left by one column.
0x0400	RTARROW	A box containing a right arrow at the right of the bottom edge of the window. If the user clicks in this box, a WM_ARROWED message is sent to the application, which should respond by scrolling the window contents right by one row.
0x0800	HSLIDE	A scroll bar and slider along the bottom edge of the window. The user can click in the bar to the right or left of the slider box, causing a WM_ARROWED message requesting a scroll of one page to be sent, or drag the slider box to a new position, causing a WM_HSLID message to be sent.

A window with all the above components will appear approximately as follows :-



The appearance may vary slightly on different versions of GEM – for example, the arrows are hollow on the Atari version, and the window sliders are the full width of the slider bar in GEM version 1.1.

The work area of a window is maintained by the application, and the application is free to output any text, VDI graphics, or AES object trees within this area. Before making any such output, the application should notify GEM AES that a window update is about to be performed using `wind_update`, as otherwise a drop down menu could still be onscreen obscuring part of the window. Note that the output is not automatically clipped to the work area, nor (when a window is partially covered by other windows) is it clipped to the visible portion of the window. When outputting to the top (active) window, the application may specify a clipping rectangle equal to the work area of the window – however, the top window can change without warning, and the application must assume after any `evnt_mesag` or `evnt_multi` call that it may have done so.

If an application is outputting to an underlying window, or (more commonly) redrawing it as a result of a redraw message, the process required is more complicated, as the application must output to each visible rectangle separately. To assist in this, the GEM AES maintains for each window a list of rectangles which are visible. The top rectangle will always have a single rectangle corresponding to the portion of the work area which is not off screen, while a window which is partially covered by seven other windows will have a rather more fragmented visible area, and therefore a longer rectangle list. A window which is completely obscured will have no rectangles in its visible list. When an application receives a redraw message, or wishes to redraw part or all of an underlying window, it should proceed as follows:

1. Notify GEM AES that it is about to update a window, using `wind_update` with a parameter value of BEG_UPDATE (see section 11.8).

2. Obtain the first rectangle in the window's list, using `wind_get` with a value of WF_FIRSTXYWH in the `w_field` parameter (see section 11.5).

3. If this rectangle's width and height are not zero (this is used to indicate the end of the rectangle list), obtain the intersection of the rectangle with the area to be redrawn. Otherwise go to step 6.

4. If the two areas intersect, set the clipping rectangle to this intersection, and output the window contents to it.

5. Obtain the next rectangle in the visible list using `wind_get` with a value of WF_NEXTXYWH in the `w_field` parameter, and go to step 3.

6. Notify GEM AES that the window update is over, using `wind_update` with a parameter value of END_UPDATE.

11.1 Create Window `wind_create`

Create Window is used to obtain a window handle to identify a window for future GEM AES window library calls. GEM AES will allocate the space in which it stores the information about the window, but does not cause the window to be displayed on the screen – see `wind_open` in section 11.2.

11.1.1 Definition

The Prospero C definition of Create Window is :

```
WORD wind_create(WORD kind,
                 WORD wx, WORD wy, WORD ww, WORD wh);
```

11.1.2 Purpose

This function is used by an application to obtain the window handle to a new window, and must be used prior to performing any operation on a window. GEM AES supports a maximum of 8 windows, though as some of these may be used by desk accessories it is not safe to assume that all 8 are available, nor is it desirable to use all 8 or desk accessories will be unable to run. The application must specify the coordinates and size of the window's maximum size – this is the rectangle returned when an application inquires what a window's full size is – see `wind_get` (section 11.5) – but does not appear to have any other purpose. However it would be sensible for an application to treat this as a limit on the maximum size that the window can assume.

11.1.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`kind`	*WORD*	**Window components** Each bit of the parameter indicates whether or not a particular window feature is present, as described in the introduction to section 11. The bit values (defined in AESBIND.H) should be combined using the OR (`|`) operator to indicate which features are required. If a particular combination is used frequently, a macro could be declared.

wx	*WORD*	**Full size X coordinate**
wy	*WORD*	**Full size Y coordinate**
ww	*WORD*	**Full size width**
wh	*WORD*	**Full size height**

The coordinates and size of the full size window, in pixels. These indicate the maximum size that the window will be allowed to take, and will typically correspond to the work area of the desktop surface.

11.1.4 Function Result

The value returned is the window's handle, which must be passed as the first parameter to all subsequent window library calls to identify the window. Valid handles are in the range 1 to 8; handle 0 is used to refer to the desktop window. A negative value indicates that no window could be allocated, normally because all 8 window handles are in use. This may be due to a badly behaved application which has not deleted windows using `wind_delete` (section 11.4) when it has finished with them.

11.1.5 Example

```
#define WF_WXYWH 4
#define NAME     0x0001
#define CLOSE    0x0002
            /* All these are in AESBIND.H */

WORD my_window;
WORD x, y, w, h;

 /* Get size of desktop work area - see section 11.5 */
  wind_get(0, WF_WXYWH, &x, &y, &w, &h);

  my_window = wind_create(NAME | CLOSE, x, y, w, h);

  if (my_window > 0)
    { /* Successfully opened */
      ...
    }
```

11.2 Open Window `wind_open`

Open Window is used to open a previously created window, causing GEM AES to draw the border area (slider bars, title etc.) on the screen.

11.2.1 Definition

The Prospero C definition of Open Window is :

```
WORD wind_open(WORD handle,
               WORD wx, WORD wy, WORD ww, WORD wh);
```

11.2.2 Purpose

This function is used by an application to open a window and display it on the screen. The window opened will be made the top (active) window. GEM AES will draw the border areas of the window; the application should then draw the window's contents, or fill the work area with a white background if the window is empty. If the window was created with the NAME or INFO bits set, indicating that is has a title or an information line,then the text of these must have been set up either using `wind_set` (section 11.6) or using the additional Prospero C bindings `wind_title` and `wind_info` (section 11.10) before the window is opened, otherwise garbage may be displayed, or GEM may crash.

11.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`handle`	*WORD*	**Window handle** The handle of the window to be opened, as returned by `wind_create` when the window was created.
`wx`	*WORD*	**Window X coordinate**
`wy`	*WORD*	**Window Y coordinate**
`ww`	*WORD*	**Window width**
`wh`	*WORD*	**Window height** The coordinates and size (in pixels) with which the window is to be opened.

11.2.4 Function Result

This function returns a value which will be zero if an error occurs, or greater than zero if no error is detected.

11.2.5 Example

```
#define WF_WXYWH 4
#define NAME     0x0001
#define CLOSE    0x0002
                    /* All these are in AESBIND.H */

WORD my_window;
WORD fx, fy, fw, fh;  /* Full size area     */
WORD x, y, w, h;      /* Current window area*/

 /* Get size of desktop work area - see section 11.5 */
 wind_get(0, WF_WXYWH, &fx, &fy, &fw, &fh);

 my_window = wind_create(NAME | CLOSE,
                         fx, fy, fw, fh);
 if (my_window > 0)
   { x = fw/4 + fx;
     y = fh/4 + fy;
     w = fw/2;
     h = fh/2;

     /* Must set title before opening */
     wind_title(my_window, "New window");

     /* Open window, quarter size and central */
     wind_open(my_window, x, y, w, h);
     ...
   }
```

11.3 Close Window `wind_close`

Close Window is used to close a previously opened window, causing it to be removed from the screen, and any areas it covered to be redrawn either by the GEM AES screen handler for areas of the desktop background, or by the owner of any windows which were partially or completely covered, by sending them redraw messages.

11.3.1 Definition

The Prospero C definition of Close Window is :

```
WORD wind_close(WORD handle);
```

11.3.2 Purpose

This function is used by an application to close a window and redraw the area of the screen it covered. The window handle is not made available for re-use, and indeed the window can be reopened using `wind_open` (section 11.3) without having to recreate it. An application should ensure that all windows are closed and deleted (in that order) before it terminates.

11.3.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`handle`	*WORD*	**Window handle** The handle of the window to be closed, as returned by `wind_create` (section 11.1) when the window was created.

11.3.4 Function Result

This function returns zero if an error occurred, or greater than zero if no error was detected.

11.3.5 Example

```
#include <aesbind.h>

WORD my_window;
WORD buffer[8];

  evnt_mesag(buffer);  /* Await a message */
  switch (buffer[0]) {
    case WM_CLOSED: /* User clicked close box, so do */
                    wind_close(buffer[3]);
                    break;
    ...
```

11.4 Delete Window `wind_delete`

Delete Window is used to free the space used by GEM AES for a window, and to make the window handle available for reuse. The window must be closed before this function is used, or the image of the window will not be removed from the screen.

11.4.1 Definition

The Prospero C definition of Delete Window is :

```
WORD wind_delete(WORD handle);
```

11.4.2 Purpose

This function is used by an application when it no longer requires a window handle, and wants to make it available for reuse. No window library calls should be made using this window handle after this function has been called. An application should ensure that all windows are closed and deleted (in that order) before it terminates, or subsequent applications will find that less than 8 windows are available, and the GEM screen manager may continue to draw windows whose application has terminated.

11.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`handle`	*WORD*	**Window handle** The handle of the window to be deleted, as returned by `wind_create` (section 11.1) when the window was created.

11.4.4 Function Result

This function returns zero if an error occurs and greater than zero if no error occurs.

11.4.5 Example

```
WORD my_windows[4];
int  i;
.
.
  /* About to terminate ... */
  /* Close then delete any windows I was using */
  for (i = 0; i < 4; i++)
    if (my_windows[i] > 0)
      { wind_close (my_windows[i]);
        wind_delete(my_windows[i]);
      }
  appl_exit();
```

11.5 Inquire Window Attributes `wind_get`

Inquire Window Attributes is used to return a variety of information about a window, depending upon the value of the `w_field` parameter.

11.5.1 Definition

The Prospero C definition of Inquire Window Attributes is :

```
WORD wind_get(WORD w_handle, WORD w_field, WORD *pw1,
              WORD *pw2, WORD *pw3, WORD *pw4);
```

11.5.2 Purpose

This function is used by an application to discover the values of a number of the fields in the internal window information maintained by GEM AES. This includes various size information, information about the slider positions, and the window's rectangle list (see the introduction to section 11). The meanings of the values returned via the parameters `pw1`, `pw2`, `pw3`, and `pw4` depend upon the value of the `w_field` parameter.

The possible values of `w_field` are as follows:

4 (WF_WXYWH)	Return the coordinates and size of the window's work area.
5 (WF_CXYWH)	Return the coordinates and size of the entire window, including border and shadow.
6 (WF_PXYWH)	Return the previous coordinates and size of the entire window. This might be used when a user clicks in the full box of a full size window, indicating that the window is to be returned to its previous size.
7 (WF_FXYWH)	Return the coordinates and size of the full size window, including border and shadow, as specified when the window was created (see section 11.1). An application might use this when the user clicks in the full box, to indicate that the window is to be made full size.

8 (WF_HSLIDE) Return a number in the range 1 to 1000 in `*pw1`, giving the position of the horizontal slider. A value of 1 means at the left hand end, while 1000 means at the right.

9 (WF_VSLIDE) Return a number in the range 1 to 1000 in `*pw1`, giving the position of the vertical slider. A value of 1 means at the top, while 1000 means at the bottom.

10 (WF_TOP) Return the window handle of the top (active) window in `*pw1`.

11 (WF_FIRSTXYWH) Return the coordinates and size of the first rectangle in the window's rectangle list (see the introduction to section 11).

12 (WF_NEXTXYWH) Return the coordinates and size of the next rectangle in the window's rectangle list (see the introduction to section 11).

15 (WF_HSLSIZE) Return a number in the range 1 to 1000 in `*pw1`, giving the size of the horizontal slider relative to the slider bar. A value of –1 may be returned to indicate the default minimum size (a square box).

16 (WF_VSLSIZE) Return a number in the range 1 to 1000 in `*pw1`, giving the size of the vertical slider relative to the slider bar. A value of –1 may be returned to indicate the default minimum size (a square box).

17 (WF_SCREEN) Return the address and size of the internal menu/alert buffer as follows:

`*pw1` – low word of address
`*pw2` – high word of address
`*pw3` – low word of length
`*pw4` – high word of length

The buffer may be used by an application for storing images of portions of the screen only if there is no possibility of a menu dropping down or alert being made while the buffer is in use. The buffer is large enough to hold a quarter of the screen image.

11.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
w_handle	*WORD*	**Window handle** The handle of the window about which the application is inquiring, as returned by wind_create (section 11.1) when the window was created. When the value of w_field is 10 (WF_TOP) or 17 (WF_SCREEN) the value of this parameter is not used.
w_field	*WORD*	**Information required** The information which the application wants returned. See above for the possible values.
pw1	*WORD* *	**First information word**
pw2	*WORD* *	**Second information word**
pw3	*WORD* *	**Third information word**
pw4	*WORD* *	**Fourth information word** These parameters point to objects which return the requested information. The meanings of the values returned depend on the value of w_field. Where the values returned are the coordinates and size of a rectangle, the values will be returned as follows: *pw1 – X coordinate of rectangle *pw2 – Y coordinate of rectangle *pw3 – Width of rectangle *pw4 – Height of rectangle Where a single value is returned, it is returned in *pw1.

11.5.4 Function Result

This function returns zero if an error occurs and greater than zero if no error occurs.

11.5.5 Example

```
#include <aesbind.h>

WORD my_window;
WORD x, y, w, h;

  /* rx, ry, rw, rh contain redraw rectangle from AES*/

  /* Get first visible rectangle of window */
  wind_get(my_window, WF_FIRSTXYWH, &x, &y, &w, &h);

  while ( (w != 0) && (h != 0) )
    { /* Do this for each valid rectangle in list */
      if (intersect(x, y, w, h, rx, ry, rw, rh))
        /* Function intersect must be provided! */
        /* Redraw area of intersection */;

      /* Get next visible rectangle of window */
      wind_get(my_window, WF_NEXTXYWH, &x, &y, &w, &h);
    }
```

11.6 Set Window Attributes `wind_set`

Set Window Attributes is used to set a variety of information about a window, depending upon the value of the `w_field` parameter.

11.6.1 Definition

The Prospero C definition of Set Window Attributes is :

```
WORD wind_set(WORD w_handle, WORD w_field,
              WORD w1, WORD w2, WORD w3, WORD w4);
```

11.6.2 Purpose

This function is used by an application to set the values of a number of the fields in the internal window information maintained by GEM AES. This includes various size information, information about the slider positions, and the window's update list (see the introduction to this section). The meanings of the parameters `w1`, `w2`, `w3`, and `w4` depend upon the value of the `w_field` parameter.

The possible values of `w_field` are as follows:

2 (WF_NAME) Set the address of the title string. The parameters `w1` and `w2` should contain the first and second words of the address of a null-terminated string. Note that GEM AES remembers the address of the string rather than making a copy of it, and therefore the string whose address is passed must remain in existence for as long as the window is displayed. A string constant is particularly suitable. Note also that the title must be set before the window is opened if the window was created with a title bar. See the Prospero C binding `wind_title` in section 11.10, which provides a much simpler interface to achieve this.

3 (WF_INFO) Set the address of the information string. The same comments apply as for WF_NAME above. See the Prospero C binding `wind_info` in section 11.10, which provides a much simpler interface.

5 (WF_CXYWH) Set the coordinates and size of the entire window, including border and shadow. This may cause a redraw message to be issued for this or other windows.

8 (WF_HSLIDE) Set the position of the horizontal slider. Pass a number in the range 1 to 1000 in `w1`, where a value of 1 means at the left hand end, and 1000 means at the right.

9 (WF_VSLIDE) Set the position of the vertical slider. Pass a number in the range 1 to 1000 in `w1`, where a value of 1 means at the top, and 1000 means at the bottom.

10 (WF_TOP) Make the window whose handle is passed in `w1` the top (active) window.

14 (WF_NEWDESK) Pass the address (first word in `w1`, second word in `w2`) of an object tree which is to be drawn as the desktop background. The first object to draw in the tree should be passed in `w3`. The object tree should completely cover the desktop area or parts of the screen will be left with garbage when a window is moved. This function should be used with great caution – no method is provided to restore normal desktop redrawing when an application terminates, so that unless the next application run is the GEM Desktop (which has privileged information on how to do this), the new desktop surface will continue to be redrawn even after the application terminates. By this time the memory containing the object tree being drawn will have been returned to the operating system, and will therefore become corrupted some random interval later, causing the next application to crash. See the Prospero C binding `wind_newdesk` in section 11.11, which provides a simpler interface to avoid having to calculate the low and high words of the address.

15 (WF_HSLSIZE) Set the size of the horizontal slider relative to the slider bar. Pass a number in the range 1 to 1000 in `w1`, or a value of –1 to indicate the default minimum size (a square box).

16 (WF_VSLSIZE) Set the size of the vertical slider relative to the slider bar. Pass a number in the range 1 to 1000 in `w1`, or a value of –1 to indicate the default minimum size (a square box).

18 (WF_TATTRB) Set the window attribute bit vector. In GEM version 2.0, this consists of a single bit (the least significant bit) indicating whether the window is on top or not – other bits are reserved for future use and should be zero. This is not provided in GEM version 1.1.

19 (WF_SIZTOP) Move the window whose handle is `w_handle` to the top, and set its coordinates and size to the values in `w1` to `w4`.

This is not provided in GEM version 1.1

Where the coordinates and size of a rectangle are being set, `w1` gives the x coordinate, `w2` the y coordinate, `w3` the width and `w4` the height.

11.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`w_handle`	*WORD*	**Window handle** The handle of the window which the application is modifying, as returned by `wind_create` (section 11.1) when the window was created. When the value of `w_field` is 10 (WF_TOP) or 14 (WF_NEWDESK) the value of this parameter is not used.
`w_field`	*WORD*	**Information to set** The information which the application wants to set. See above for the possible values.
`w1`	*WORD*	**First information word**
`w2`	*WORD*	**Second information word**
`w3`	*WORD*	**Third information word**
`w4`	*WORD*	**Fourth information word** The meanings of these parameters depend on the value of `w_field`. Where the values set are the coordinates and size of a rectangle, the values are passed as follows: `w1` – X coordinate of rectangle `w2` – Y coordinate of rectangle `w3` – Width of rectangle `w4` – Height of rectangle Where a single value is set, it is passed in `w1`.

11.6.4 Function Result

This function returns zero if an error occurs, and greater than zero if no error occurs.

11.6.5 Example

```
#include <aesbind.h>

WORD my_window;
WORD x, y, w, h;

  /* Find full size of window, and set it */

  wind_get(my_window, WF_FXYWH, &x, &y, &w, &h);
  wind_set(my_window, WF_CXYWH, x, y, w, h);
```

11.7 Find Window Under Point `wind_find`

Find Window Under Point is used to discover which window lies under a certain point, usually the mouse position. Windows will be checked from the top window and working backwards.

11.7.1 Definition

The Prospero C definition of Find Window Under Point is :

```
WORD wind_find(WORD mx, WORD my);
```

11.7.2 Purpose

This function is used by an application to find out which window lies under the specified point. The most common use of this is to discover the window under the current mouse position. Windows are searched from the front window backwards, which has the effect that the window which is visible at the specified point will be found. If no window lies under the specified point, a value of zero will be returned – this is the handle of the desktop surface.

11.7.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
mx	*WORD*	**X coordinate of point**
my	*WORD*	**Y coordinate of point**
		The coordinates of the point whose corresponding window is to be found.

11.7.4 Function Result

This function returns the window handle of the window which is visible at the specified point, or zero if the desktop surface is visible.

11.7.5 Example

```
WORD my_window;
WORD dummy;
WORD mx, my;

  /* wait for a click */
  evnt_button(1, 1, 1, &mx, &my, &dummy, &dummy);
  if (wind_find(mx, my) != my_window)
    /* clicked outside window - beep */
  else
    { /* Do something with click */
      ...
    }
```

11.8 Start or End Window Update `wind_update`

Start or End Window Update is used to indicate that an application is about to start or finish updating a window, or that it wants to take or relinquish control of mouse functions.

11.8.1 Definition

The Prospero C definition of Start or End Window Update is :

```
WORD wind_update(WORD begend);
```

11.8.2 Purpose

This function is used by an application to prevent drop down menus from appearing or other windows from being updated while it is outputting to a window. If an application does not signal the start of an update before outputting to a window, it has no way of knowing whether the portion of the window it is outputting to is covered by a menu, or whether the user is holding down the mouse button half way through a window move operation, for example. When the application indicates that it is entering window update mode, GEM AES will wait until all menus are put away and no drags of windows, sliders etc. are in progress before returning. It is standard practice to enter update mode as soon as an event occurs before processing it, and to leave update mode immediately before waiting for the next event. Care should be taken that the application leaves update mode before terminating, or the GEM Desktop (or whatever application runs next) will not function, and the machine may have to be rebooted.

This function also allows an application to specify whether the mouse is to interact with the menu and the window control points or not. This would be used for example by an application doing its own form processing rather than using `form_do` (section 7.1), to prevent menus from appearing when the mouse was moved into the menu bar. Once again, normal mouse function must be restored before terminating, and before any message event is waited for.

11.8.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
begend	*WORD*	**Update begin or end** A value in the range 0 to 3 as follows: 0 (END_UPDATE) Leave window update mode 1 (BEG_UPDATE) Enter window update mode 2 (END_MCTRL) Leave mouse control mode 3 (BEG_MCTRL) Enter mouse control mode The above macros are defined in AESBIND.H.

11.8.4 Function Result

This function returns zero if an error occurs and greater than zero if no error occurs.

11.8.5 Example

```
#include <aesbind.h>

WORD event;

  do {
    event = evnt_multi /*lots of parameters here*/;
    wind_update(BEG_UPDATE);
    /* Process the event */
    wind_update(END_UPDATE);
  } while /* the event did not cause the application to
             terminate */
```

11.9 Calculate Window Coordinates `wind_calc`

Calculate Window Coordinates is used to convert between the coordinates and size of an entire window (including the border area) and those of the window's work area, or vice versa.

11.9.1 Definition

The Prospero C definition of Calculate Window Coordinates is :

```
WORD wind_calc(WORD workflag, WORD kind,
               WORD x, WORD y, WORD w, WORD h,
               WORD *px, WORD *py, WORD *pw, WORD *ph);
```

11.9.2 Purpose

This function is used by an application to calculate the coordinates of a window's work area given the coordinates of the entire window, or vice versa. This might be used for example when a window has been moved – the new coordinates of the entire window are specified in the WM_MOVED message from GEM AES, but when redrawing the contents of the window, the application will need to know the coordinates of the window's work area. The application must specify what features the window has, as this affects the size of the border area; this is done using a bitmap in the same form as for `wind_create` (section 11.1) in the parameter `kind`.

11.9.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`workflag`	*WORD*	**Return work area flag** This indicates whether the function is to return the coordinates and size of the work area (`workflag` = 1) or of the entire window (`workflag` = 0).
`kind`	*WORD*	**Window features** A bitmap specifying the features present in the window, in the same format as for the parameter `kind` in `wind_create`. See the introduction to section 11 for further details.
`x`	*WORD*	**Input X coordinate**
`y`	*WORD*	**Input Y coordinate**
`w`	*WORD*	**Input width**
`h`	*WORD*	**Input height** The coordinates and size of the work area or entire window, from which the coordinates and size of the entire window or work area are to be calculated.
`px`	*WORD* *	**Output X coordinate**
`py`	*WORD* *	**Output Y coordinate**
`pw`	*WORD* *	**Output width**
`ph`	*WORD* *	**Output height** These parameters point to objects which return the coordinates and size of the work area or entire window, calculated from the values in `x`, `y`, `w` and `h`.

11.9.4 Function Result

This function returns zero if an error occurs and greater than zero if no error occurs.

11.9.5 Example

```
#include <aesbind.h>

#define my_features 0x0fee
                        /* All but info line and name*/

WORD my_window;
WORD x, y, w, h;
WORD wx, wy, ww, wh;

  /* Get desktop area */
  wind_get(0, WF_WXYWH, &x, &y, &w, &h);

  my_window = wind_create(my_features, x, y, w, h);
  if (my_window > 0)
    {
    /* Open a full size window */
    wind_open(my_window, x, y, w, h);

    /* Get coordinates of work area */
    wind_calc(1, my_features, x, y, w, h,
              &wx, &wy, &ww, &wh);

    /* Output to the work area */
    .
    .
```

11.10 Set Window Title or Info

`wind_title`
`wind_info`

Set Window Title and Set Window Info are provided by the Prospero C bindings as alternatives to the use of `wind_set` (section 11.6), to avoid the need to calculate the low and high order words of the string's address. These functions are not provided by the original Digital Research bindings.

11.10.1 Definition

The Prospero C definitions of Set Window Title and Info are :

```
WORD wind_title(WORD handle, const char *title);

WORD wind_info(WORD handle, const char *info);
```

11.10.2 Purpose

These functions may be used by an application to set the title or information line of a window created with the NAME or INFO attributes (see section 11.1), to provide a simpler interface than the use of `wind_set` (section 11.6). Any window created with either the NAME or INFO attribute must have the title or information line initialized either using one of these functions or using `wind_set` before the window is opened, or garbage will be displayed and a system crash may occur.

As GEM AES stores the address of the title or information string rather than a copy of its contents, the string passed must remain in existence (and unmodified) for the entire time that the window is displayed. A string literal is therefore particularly suitable.

11.10.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
handle	*WORD*	**Window handle** The handle of the window whose title or information line is being set.
title	*const char **	**New window title**
info	*const char **	**New window information** These parameters point to null-terminated strings containing the new text to be used for the window title or to be displayed on the window's information line.

11.10.4 Function Result

These functions both cause the GEM AES function `wind_set` (section 11.6) to be called, and the value returned reflects the result of that operation. The value returned will be zero if an error occurs and greater than zero if no error occurs.

11.10.5 Example

```
WORD my_window;
WORD x, y, w, h;
WORD insert_mode;

  /* Create window with title, info line
    (and all other features) */
  my_window = wind_create(0x0fff, x, y, w, h);

  /* Set up title and info before opening */
  wind_title(my_window, "Work window");
  wind_info(my_window, "Insert mode");

  insert_mode = 1;

  wind_open(my_window);

  /* Later ... */
  /* Update info line to indicate state of flag*/
  wind_info(my_window, insert_mode ? "Insert mode" :
                                     "Overwrite mode");
```

11.11 Set New Desktop `wind_newdesk`

Set New Desktop is provided by the Prospero C bindings as an alternative to the use of `wind_set` (section 11.6), to avoid the need to calculate the low and high order words of the form's address. This function is not provided by the original Digital Research bindings.

11.11.1 Definition

The Prospero C definition of Set New Desktop is :

```
WORD wind_newdesk(OBJECT *newdesk, WORD firstobj);
```

11.11.2 Purpose

This function may be used by an application to specify the address of an object which is to be drawn as the desktop background on those areas of the screen not covered by windows. This form may contain icons and so on which the user can select, and so on.

Any application which specifies a new tree for the desktop is likely to be dangerous for other applications to execute using the Prospero C `spawn...` functions, as there is no way to reset the desktop to its previous tree, and the memory containing the form being drawn will be released to the operating system when the application which set the new desktop terminates. When this memory is reused (probably some time after the application terminated) the currently running application will crash. It is therefore only safe to redefine the desktop if you are certain that any program which might execute your application also does so, and will reset it as soon as your application terminates.

11.11.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`newdesk`	*OBJECT* *	**New desktop background tree** The tree which is to be drawn as the new desktop background.
`firstobj`	*WORD*	**First object to draw** The first object to draw in the tree.

11.11.4 Function Result

This function causes the GEM AES function `wind_set` to be called, and the value returned reflects the result of that operation. A value of zero indicates an error has occurred and a value greater than zero means no error has occurred.

11.11.5 Example

```
OBJECT * my_desk;
char * child;

  spawnl(P_WAIT, child, NULL);
  /* Reset to my desktop in case child set its own */
  wind_newdesk(my_desk, 0);
```

12 RESOURCE LIBRARY

This section contains descriptions of the Resource Library functions, in the following sub-sections.

Section	Function description	Binding name
12.1	Load Resource File	`rsrc_load`
12.2	Free Resource File Memory	`rsrc_free`
12.3	Get Resource Address	`rsrc_gaddr`
12.4	Store Resource Address	`rsrc_saddr`
12.5	Convert Object Coordinates	`rsrc_obfix`

These functions are concerned with the management of resource files and their contents. All applications normally have an associated resource file, with the same name but extension .RSC, containing the object trees that define the menu bar, any dialog boxes used, and possibly other data such as icons, bit images and so on. The resource file is created using a resource editor, which also provides an include file which defines the constants used to refer to the various structures and objects in the file. The application can then use `rsrc_load` (section 12.1) to load the data in the resource file into memory and prepare it for use by the application, and `rsrc_gaddr` (section 12.3) to obtain the addresses of the data structures thus loaded. These can then be displayed or manipulated in the normal way using the functions described in sections 6 and 7.

The advantages of using a resource file rather than creating the object trees dynamically within the application source are several. Firstly, the code to produce an object tree dynamically is complicated and obscure, and requires a thorough understanding of the tree structure. An application which includes the code to produce its own menu tree dynamically is likely to be larger and will take longer to write than the time and space required to create an application and a resource file. Secondly, many alterations to the resource file may not require the application to be recompiled – this is of particular advantage when customizing an application for a foreign language. Thirdly, an application which produces its object trees dynamically is unlikely to be machine independent, as it is hard to cope with the possibility of different screen resolutions (handled easily using resource files) and there are instances where the different word order used by the Motorola 68000 and Intel 8086 processors affect the way in which object trees must be coded. By simply recreating the resource file in a different environment, this problem can be avoided.

To create a resource file using a resource editor is simple and quick, allowing more time to go into careful design of the form layouts; when creating an object tree dynamically, the process is slow and tricky to write, and any changes to the layout of forms will be painful.

Note that most resource editors have an option whereby the resource information can be generated in the form of C initialized static data declarations, which give a third possible method of creating object trees. This would have most of the advantages of using a resource file (although changes would always require recompilation of the static data), with the added advantage that only one file needs to be present for the application to be executed.

12.1 Load Resource File `rsrc_load`

Load Resource File is used to allocate memory and load an application's resource file, containing such things as the menu, dialog forms and so on.

12.1.1 Definition

The Prospero C definition of Load Resource File is :

```
WORD rsrc_load(const char *rsname);
```

12.1.2 Purpose

This function is used by an application to load into memory the contents of its resource file, and must be used before any other resource library calls can be made. This function also causes all objects in the file to be converted for the screen resolution in use. Note that only one resource file can be loaded at a time.

12.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`rsname`	*const char **	**Resource file name** The filename (and optional path specifier) of the resource file. All resource files have the extension .RSC, and by convention the same name as the application to which they belong. As they will reside in the same directory as the application, which will usually be the default directory when an application is running, the path specifier may not be needed. However a more robust application might use `shel_find` (section 13.3) to locate its resource file before loading it.

12.1.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected. It is prudent to check this value, and terminate (after notifying the user of the problem) if the value is zero, indicating that the resource file cannot be found.

12.1.5 Example

```
#include <aesbind.h>

main()
{ appl_init();
  if (rsrc_load("MYPROG.RSC") == 0)
    form_alert(1,"[1][No Resource File!][OK]");
  else
    { /* main body of program in here */
      ...
      rsrc_free();
    }
  appl_exit();
}
```

12.2 Free Resource File Memory `rsrc_free`

Free Resource File Memory is used to free the memory allocated by `rsrc_load`.

12.2.1 Definition

The Prospero C definition of Free Resource File Memory is :

```
WORD rsrc_free(void);
```

12.2.2 Purpose

This function is used by an application to free the memory containing the resources. This must not be used while any of the resources are in use, such as the menu bar. This function is most commonly used when an application is about to terminate.

12.2.3 Parameters

There are no parameters.

12.2.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected.

12.2.5 Example

See section 12.1.5.

12.3 Get Resource Address `rsrc_gaddr`

Get Resource Address is used to obtain the address of a data structure in a resource file loaded into memory using `rsrc_load`.

12.3.1 Definition

The Prospero C definition of Get Resource Address is :

```
WORD rsrc_gaddr(WORD rstype, WORD rsid,
                OBJECT * *paddr);
```

12.3.2 Purpose

This function is used by an application to obtain the address of a data structure from the resource file, so that it can be used by the application. The application must have loaded the resource file using `rsrc_load` (section 12.1) before using this function.

The application must supply the index and type of the structure whose address it wants to obtain. The index will be supplied as a macro in the header file generated by the resource editor, and should be passed in the parameter `rsid`. The type is specified using the parameter `rstype`, and should be one of the following:

0	object tree
1	object
2	`tedinfo` structure
3	`iconblock` structure
4	`bitblk` structure
5	null-terminated string
6	`imagedata`
7	`ob_spec`
8	`te_ptext`
9	`te_ptmplt`
10	`te_pvalid`
11	`ib_pmask`
12	`ib_pdata`
13	`ib_ptext`
14	`bi_pdata`
15	`ad_frstr` (the address of a pointer to a free string)
16	`ad_frimg` (the address of a pointer to a free image)

See section 6 for more information on these data structures.

12.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
rstype	*WORD*	**Resource type** The type of the resource whose address is to be returned, as listed above. To load a menu or dialog tree, use the value zero.
rsid	*WORD*	**Resource index** The index of the resource whose address is required. This will be a macro provided by the resource editor when the resource file was created.
paddr	*OBJECT* * *	**Resource address** This parameter points to an object which receives a pointer to the specified data structure. In order to pass a parameter of a different type (as required when using a value other than zero as the value of the rstype parameter) without generating a warning, a cast should be employed.

12.3.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected.

12.3.5 Example

```
#define menu 0          /* Provided by resource editor */

OBJECT *TheMenu;


  rsrc_load("MYPROG");           /* Load resource file */
  rsrc_gaddr(0, menu, &TheMenu);
                                 /* Get address of menu */
  menu_bar(TheMenu, 1);           /* .. and display it */
```

12.4 Store Resource Address `rsrc_saddr`

Store Resource Address is used to store the address of a data structure into the array containing the addresses of items loaded from a resource file using `rsrc_load`.

12.4.1 Definition

The Prospero C definition of Store Resource Address is :

```
WORD rsrc_saddr(WORD rstype, WORD rsid, void *lngval);
```

12.4.2 Purpose

This function is used by an application to store the address of a data structure in memory.

The application must supply the type of the structure whose address it wants to store, and the index of the data structure. The type is specified using the parameter `rstype`, and should be one of the following:

```
15  ad_frstr   /*the address of a pointer to a free string*/
16  ad_frimg   /*the address of a pointer to a free image*/
```

12.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`rstype`	*WORD*	**Resource type** The type of the resource whose address is to be stored, as listed above.
`rsid`	*WORD*	**Resource index** The location in the data structure where the address is to be stored.
`lngval`	*void **	**Resource address** This parameter contains the address to be stored.

12.4.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected.

12.4.5 Example

```
#define fstraddr 2
  /* Sample constant from resource editor */

  rsrc_saddr(15, fstraddr, "New free string");
```

12.5 Convert Object Coordinates `rsrc_obfix`

Convert Object Coordinates is used to convert the coordinate fields of an object in an object tree from the character based (resolution independent) form in which they are stored in a resource file to pixel coordinates.

12.5.1 Definition

The Prospero C definition of Convert Object Coordinates is :

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

12.5.2 Purpose

This function is used by an application to convert the coordinates of an object from the resolution independent form in which they are stored in a resource file to pixel coordinates. Note that `rsrc_load` (section 12.1) converts all coordinates when it loads the resource file, so that it is unusual for an application to require this function. However, as it is required internally by the `rsrc_load` function, there is no harm in providing it.

12.5.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`tree`	*OBJECT* *	**Tree containing object**
		The tree containing the object to be converted.
`obj`	*WORD*	**Index of object to convert**
		The index in the object tree of the object whose coordinates are to be converted.

12.5.4 Function Result

This function always returns the value one.

12.5.5 Example

```
#define obj1 20
          /* Sample constant from resource editor */

OBJECT * my_tree;

   rsrc_obfix(my_tree, obj1);
```

13 SHELL LIBRARY

This section contains descriptions of the Shell Library functions, in the following sub-sections.

Section	Function description	Binding name
13.1	Shell Read	`shel_read`
13.2	Shell Write (version 1.1) Shell Write (version 2.0)	`shel_write_1` `shel_write_2`
13.3	Shell Find	`shel_find`
13.4	Search Shell Environment	`shel_envrn`
13.5	Return Default Application	`shel_rdef`
13.6	Set Default Application	`shel_wdef`

The Shell is the name given to the parts of GEM VDI and AES that concern the execution of applications. On the Atari ST, this forms an integral part of the operating system, and a large proportion of what follows is not relevant. On 8086 and family computers, such as the IBM PC or the Amstrad 1512, the GEM environment in which GEM applications execute is loaded into memory after the underlying operating system (MS-DOS or DOS Plus), and the loading of this environment, and of the applications that run in it, is handled by the Shell manager, which is part of the GEMVDI.EXE file. The GEM Desktop Application, from which other applications can be started by double clicking or opening, is just a normal GEM application, which is normally executed within the GEM environment by the Shell Manager when GEM VDI and AES are first loaded. When an application terminates, the Shell Manager will normally execute the Desktop once again, to allow the user to select another application to execute. However, it is possible to bypass the Desktop, both when starting an application and when it terminates. If a parameter is given when starting GEM from the operating system, such as 'GEM progname', the specified application will be started after the VDI and AES are loaded, rather than the GEM Desktop, and when it terminates, the user will be returned to the operating system prompt. If a '/D' follows the name of the application, such as 'GEM progname /D', the Shell Manager will load and execute the GEM Desktop application when the named application terminates.

The functions in the Shell Library described in this section are concerned with altering the way the Shell Manager behaves when the current application terminates, and with discovering the manner in which the Shell Manager invoked the application. There are also functions to search for a file, and to return the name of the file which was opened to start an application, which will be relevant on both the IBM PC family and the Atari ST implementations of GEM.

13.1 Shell Read `shel_read`

This function allows an application to obtain the command and command tail with which the application was invoked.

13.1.1 Definition

```
WORD shel_read(char pcmd[], char ptail[]);
```

13.1.2 Purpose

This function allows an application to identify the command that invoked it, and the command tail invoked with it. If the application had been started by double clicking on its icon or name, the command tail would be empty, while if it had been installed as having a particular document type, it could be started by double-clicking on a document of that type, in which case the command tail would contain the name of the document. An application may also be started from another application by means of the Prospero C `spawn...` functions, in which case the command tail is specified by the parent application.

The GEM AES documentation states that both parameters must have space for at least 128 characters, though the command can never use more than 80 of these.

13.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
cmd	*char []*	**Command** This parameter provides the array which is used to return the system command which invoked the application – in other words the program name, possibly preceded by a path specification.
tail	*char []*	**Command tail** This parameter provides the array which is used to return the command tail passed by the system. This may contain the name (and possibly path) of a document which was opened to start the application, or have some other application dependent meaning.

13.1.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected.

13.1.5 Example

```
char cmd[129], tail[129];

  /* Get name of application and document */
  shel_read(cmd, tail);

  /* Now use them ... */
```

13.2 Shell Write

shel_write_1
shel_write_2

Shell Write is a companion function to Shell Read in that it provides facilities for GEM to execute another application after the current application terminates. It can be used to provide the command (the disk identifier, directory path and program name) as well as a command tail which can include special instructions to be passed to the new program. See section 13.1 for details of Shell Read.

The function and its binding have been altered slightly by Digital Research between versions 1.1 and 2.0 of GEM – the bindings for both versions are described below.

13.2.1 Definition

The Prospero C definitions of Shell Write are:

For GEM version 1.1

```
WORD shel_write_1(WORD doex, WORD isgr, WORD iscr,
                  const char *pcmd, const char *ptail);
```

For GEM version 2.0

```
WORD shel_write_2(WORD doex, WORD isgr, WORD isover,
                  const char *pcmd, const char *ptail);
```

13.2.2 Purpose

This function can be used to make GEM execute another application after the current one terminates, rather than return to the GEM Desktop or operating system prompt from which it was invoked. In GEM version 2.0, the application may specify that the new application is to be executed immediately rather than when the current one terminates.

13.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
doex	*WORD*	**Do execute** If this parameter is zero, no application is to be executed, so that when the current application terminates this function should cause a return to the operating system prompt. Otherwise, the parameter value should be one.
isgr	*WORD*	**Graphics application** If this parameter is 1 it indicates to GEM that the application to be executed is a graphics application. This determines whether a workstation is opened before starting it, and whether the screen will be placed in alphanumeric or graphics mode. Otherwise, the parameter value should be 0.
iscr	*WORD*	**GEM AES application** If this parameter is 1 it indicates to GEM that the application to be called is a normal GEM AES application. Otherwise the parameter value must be zero. This parameter is used in the GEM version 1.1 bindings only.
isover	*WORD*	**Overlay specifier** This parameter specifies where and when the named application is to be executed. The possible values are as follows: 0 Run application immediately in memory above current application, returning to current application when it terminates. The values of doex and isgr are ignored. This is similar to the Prospero C spawn... functions. 1 Run application when current application terminates, in the same memory as the current application. The values of doex and isgr are used.

2 Run application when current application terminates, in the memory used by the current application and GEM VDI. The values of `doex` and `isgr` are used.

This parameter is not used in the GEM version 1.1 bindings.

`pcmd` *const char ** **Command**

This parameter provides the system command which calls the application. It would normally contain the pathname of the application; that is the disk drive identifier and directory path, followed by the name of the applications program with a .PRG, .APP, .EXE etc. extension.

`ptail` *const char ** **Command tail**

This parameter provides the command tail which is passed by the system or parent application to the specified application.

13.2.4 Function Result

This function returns zero if an error occurs, or greater than zero if no error is detected.

13.2.5 Example

```
char * program_name;

  /* run a GEM AES graphics program after
    termination of current program, no command tail */

  /* GEM 1.1 version */
  shel_write_1(1, 1, 1, program_name, "");

  /* GEM 2.0 version */
  shel_write_2(1, 1, 1, program_name, "");
```

13.3 Shell Find `shel_find`

Shell Find is a useful function that searches the current DOS search path for a particular filename, and if present returns the complete pathname, including disk drive identifier, directory path name and file name.

13.3.1 Definition

The Prospero C definition of Shell Find is :

```
WORD shel_find(char ppath[]);
```

13.3.2 Purpose

This function searches for a file in the current directory and all directories in the DOS search path. If it is found, the full file specification including drive and path name is returned. Note that the array `ppath` is used for both input and output.

13.3.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`ppath`	*char []*	**File name and path** This array should be set up to contain the name and extension of a file for which the application wants to search. The function will search for this file in every directory in the current DOS search path, and also in the root, GEMDESK, GEMSYS and GEMAPPS directories. If the file is found, the complete file specification including drive and pathname is returned in the same parameter.

13.3.4 Function Result

This function returns zero if an error occurs, usually indicating that the file is not found and the contents of `ppath` are not valid. A value greater than zero indicates that there is no error, and the file is found.

13.3.5 Example

```
char filename[80];

  strcpy (filename, "MYNAME.RSC");
  if (shel_find(filename))
    rsrc_load(filename);
  else
    { form_alert(1,"[1][No Resource File][OK]");
      exit(3);
    }
```

13.4 Search Shell Environment `shel_envrn`

The DOS Environment is a message area in which programs may "pin up notices" and also scan to see what goes. It includes information as to the current command processor, the current path, prompt, and any other information which the operating system or applications may place there. The format includes an =, for example the line `COMSPEC=C:\COMMAND.COM` tells MS-DOS where to load the command processor from on termination of a program which removed it from memory (the current environment may be displayed by typing `SET` at the MS-DOS prompt). Search Shell Environment is a useful function that searches the Environment for a particular parameter string, for example "`PATH=`", and if present a copy of the string corresponding to that parameter is returned.

13.4.1 Definition

The Prospero C definition of Search Shell Environment is :

```
WORD shel_envrn(char pvalue[], const char *psrch);
```

13.4.2 Purpose

This function allows an application to search the Environment for a particular parameter. The parameter string for which the application is searching (including the '=') is passed in the parameter `psrch`. The string immediately following the '=' will be returned in the array `pvalue`. This will be an empty string if the value was not found. See also the Prospero C function `getenv`.

13.4.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
pvalue	*char []*	**Parameter value found** The string following the equals sign of the requested parameter will be returned in this string. No length checking is performed, and if the result is longer that the string used to hold it, the application may crash. If the parameter is not found, a null string will be returned.
psrch	*const char **	**Parameter to search for** Points to a string specifying the parameter in the environment string for which the application is searching, including the '=' sign.

13.4.4 Function Result

This function always returns one.

13.4.5 Example

```
char path[80];

  shel_envrn(path, "PATH=");
  if (path[0] != '\0')
    {
    /* There is a search path set up - use it */
```

13.5 Return Default Application `shel_rdef`

Return Default Application allows an application to discover the name and path specification of the application which will run when it terminates – normally this will be the GEM Desktop application.

This function is not provided in GEM version 1.1.

13.5.1 Definition

The Prospero C definition of Return Default Application is :

```
void shel_rdef(char lpcmd[], char lpdir[]);
```

13.5.2 Purpose

This function returns the name and directory of the default application, which is the one which will execute when the current application terminates, unless `shel_write` (section 13.2) has been used to specify that another application is to be run. The default application is usually the GEM Desktop application, though this may be altered using `shel_wdef` (section 13.6). This function is not supported in GEM version 1.1

13.5.3 Parameters

`Parameter name`	*Type of parameter*	**Parameter description** Function of parameter
`lpcmd`	*char []*	**Default application name** This array returns the name of the default application. GEM AES states that the space must be at least 32 characters, although only a maximum of 12 will be used for a file name.
`lpdir`	*char []*	**Default application path** This array returns the directory path of the default application.

13.5.4 Function Result

No value is returned.

13.5.5 Example

```
char name[33], path[33];


  shel_rdef(name, path);
  if (strcmp(name, "DESKTOP.APP") != 0)
    { /* an abnormal state of affairs */
      ...
    }
```

13.6 Set Default Application `shel_wdef`

Set Default Application allows an application to change the name and path specification of the application which will run when it terminates – unless changed this will be the GEM Desktop application.

This function is not provided in GEM version 1.1.

13.6.1 Definition

The Prospero C definition of Set Default Application is :

```
void shel_wdef(const char *lpcmd, const char *lpdir);
```

13.6.2 Purpose

This function sets the name and directory of the default application, which is the one which will execute when the current application terminates, unless `shel_write` (section 13.2) has been used to specify that another application is to be run. The default application is usually the GEM Desktop application, unless altered using this function. The current default application may be discovered using `shel_rdef` (section 13.5). This function is not supported in GEM version 1.1

13.6.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
lpcmd	*const char* *	**Default application name** This parameter points to a string containing the new name of the default application.
lpdir	*const char* *	**Default application path** This parameter points to a string containing the new directory path of the default application.

13.6.4 Function Result

No value is returned.

13.6.5 Example

```
char name[33], path[33];

  shel_rdef(name, path);
  if (strcmp (name, "DESKTOP.APP") == 0)
    /* No-one else has set it, so ... */
    shel_wdef("MYPROG.EXE","C:/");
```

14 EXTENDED GRAPHICS LIBRARY

This section contains descriptions of the Extended Graphics Library functions, in the following sub-sections.

Section	Function description	Binding name
14.1	Calculate Box Increments	`xgrf_stepcalc`
14.2	Draw XORed Boxes	`xgrf_2box`

These functions are provided in GEM version 2.0 to allow applications to create 'zoom box' effects, as the functions in the Graphics and Form libraries which used to perform this function have been removed to save space.

14.1 Calculate Box Increments `xgrf_stepcalc`

Calculate Box Increments is designed to help an application draw zoom boxes, and was provided in GEM version 2.0 to compensate for the removal of the zoom box functions `graf_growbox` and `graf_shrinkbox` (section 8.4) from GEM version 1.1.

This function is not provided (or needed) in GEM version 1.1.

14.1.1 Definition

The Prospero C definition of Calculate Box Increments is :

```
WORD xgrf_stepcalc(WORD orgw, WORD orgh,
                   WORD xc, WORD yc, WORD w, WORD h,
                   WORD *pcx, WORD *pcy, WORD *pcnt,
                   WORD *pxstep, WORD *pystep);
```

14.1.2 Purpose

This function calculates the required increments to draw a zoom box on the screen. It is not supported in GEM version 1.1

14.1.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`orgw`	*WORD*	**Initial width**
`orgh`	*WORD*	**Initial height**
		The initial width and height of the box in pixels.
`xc`	*WORD*	**Final X coordinate**
`yc`	*WORD*	**Final Y coordinate**
`w`	*WORD*	**Final width**
`h`	*WORD*	**Final height**
		The final coordinates and size of the box.

pcx	*WORD* *	**Centred X coordinate**
pcy	*WORD* *	**Centred Y coordinate**

These parameters point to objects which return the centred x and y coordinates at the end of the zoom.

pcnt	*WORD* *	**Step count**

This parameter points to an object which returns the number of boxes to be drawn for the zoom.

pxstep	*WORD* *	**X step increment**
pystep	*WORD* *	**Y step increment**

These parameters point to objects which return the amount to be added to the x and y coordinates for each step in the zoom. These parameters may be passed to the function `xgrf_2box` to cause the boxes to be drawn.

14.1.4 Function Result

This function returns zero if an error is detected, and greater than zero if no error is detected.

14.1.5 Example

```
WORD count;
WORD cx, cy, xstep, ystep;

  /* Zoom large box to small one in top left corner */
  xgrf_stepcalc(100, 100, 10, 10, 10, 10,
                &cx, &cy, &count, &xstep, &ystep);

  /* Actually draw the boxes, from big box at 20,20 */
  xgrf_2box(20, 20, 100, 100, 0, count,
            xstep, ystep, 0);
```

14.2 Draw XORed Boxes `xgrf_2box`

Draw XORed Boxes is designed to help an application draw zoom boxes, and was provided in GEM version 2.0 to compensate for the removal of the zoom box functions `graf_growbox` and `graf_shrinkbox` (section 8.4) from GEM version 1.1.

This function is not provided (or needed) in GEM version 1.1.

14.2.1 Definition

The Prospero C definition of Draw XORed Boxes is :

```
WORD xgrf_2box(WORD xc, WORD yc, WORD w, WORD h,
               WORD corners, WORD cnt,
               WORD xstep, WORD ystep, WORD doubled);
```

14.2.2 Purpose

This function draws a series of XORed boxes on the screen, to give the impression of a box expanding or shrinking to nothing. It is normally used in conjunction with `xgrf_stepcalc` (section 14.1) which provides some of the parameters. This function is not provided in GEM version 1.1.

14.2.3 Parameters

Parameter name	*Type of parameter*	**Parameter description** Function of parameter
`xc`	*WORD*	**Initial X coordinate**
`yc`	*WORD*	**Initial Y coordinate**
`w`	*WORD*	**Initial width**
`h`	*WORD*	**Initial height**
		The coordinates and size (in pixels) of the first box in the series of boxes.

`corners` *WORD* **Corners only flag**

If this is 1, only the corners of the boxes will be drawn, giving a faster zoom. Otherwise, the value 0 should be supplied.

`cnt` *WORD* **Step count**

The number of steps in the series. If `xgrf_stepcalc` has been used to calculate the increments, the value returned in `pcnt` should be used.

`xstep` *WORD* **X step increment**
`ystep` *WORD* **Y step increment**

The x and y increments of each step in the series of boxes. If `xgrf_stepcalc` has been used to calculate the increments, the values returned in `pxstep` and `pystep` should be used.

`doubled` *WORD* **Double steps flag**

If this is 1, twice the number of boxes will be drawn, giving a slower but smoother zoom. Otherwise, the value 0 should be supplied.

14.2.4 Function Result

This function returns zero if an error is detected, and greater than zero if no error is detected.

14.2.5 Example

See section 14.1.5.

15 INDEX OF FUNCTIONS

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

Prospero C AES Bindings