Lattice C 5.5

the C Compiler for your Atari ST/STE/TT Computer

Addendum

Libraries

Requires:

- ✓ Atari 520ST upwards
  (1M+ memory required)
- ✓ Disk drive
  (2 floppies or hard disk advised)
- ✓ Mouse

HiSoft
High Quality Software

Lattice C 5.5 for the Atari ST/STE/TT

By HiSoft and Lattice, Inc.

Program:
designed and programmed by HiSoft and Lattice, Inc.

Manual:
written by Alex Kiernan.

Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, March 1992 - ISBN 0 948517 57 3

Table of Contents

Introduction

This manual describes the additional header files and functions supplied as part of Lattice C 5.50, over and above those documented in Volume's II and III.

All the information is ordered by header file which includes both additional header files and additional functions or functionality for functions declared within the header file.

acc.h

The acc.h header file includes definitions to support the generation of desk accessories. It provides two functions: a macro for setting the stack size and the function which enables memory to be made available for `malloc()`.

_addheap — Add memory to the malloc heap

SYNOPSIS

```
#include <acc.h>

_addheap(ptr, size);

void *ptr;              pointer to base of heap
size_t size;            size of stack for DA in bytes
```

DESCRIPTION

The `_addheap` function is used to add memory to the local heap for use in a desk accessory. This is needed because a desk accessory may not legitimately call `Malloc` (the OS memory allocator) due to the way in which desk accessories are run.

The `ptr` parameter is the base of a heap which should be used, whilst size gives its `size` in bytes. Note that the base of heap should be word aligned in order to ensure that the memory is suitable for any purpose which `malloc()` may require.

`_addheap()` may be called as often as needed in a single program, but *must only be called once* for any particular block. Note that once the memory has been allocated to the heap manager using this call the memory may not be used for any other purpose.

EXAMPLE

```
#include <acc.h>
#include <aes.h>

void
main(void)
{
   static long heap[16384/sizeof(long)];   // 16K heap

   _addheap(heap, sizeof(heap));
   appl_init();
   ...
}
```

STACK — Set size of stack for a desk accessory

SYNOPSIS

```
#include <acc.h>

STACK(size);

size_t size;              size of stack for DA in bytes
```

DESCRIPTION

The `STACK()` macro is used to provide the necessary external definitions to change the size of a stack in a desk accessory from the default of 4Kb. It *must* be used outside any function as it includes a definition for a global variable.

The single parameter `size` specifies the number of bytes which the size of the stack is to be set to.

Note that there is no way in which a desk accessories stack may be changed at runtime.

EXAMPLE

```
#include <acc.h>
…

STACK(16384);   // set the stack to 16Kb

void
main(void)
{
   …
}
```

aes.h

aes.h is the AES interface file. Within this there is one additional function for 5.5, `rc_center()`, and additional functionality for TT and MegaST[E] TOS via the `WF_COLOR` and `WF_DCOLOR` `wind_set` operations.

rc_center — Centre one rectangle within another

SYNOPSIS

```
#include <aes.h>

rc_center(rect1,rect2);

const GRECT *rect1;        source rectangle
GRECT *rect2;              the target rectangle
```

DESCRIPTION

This function is used to centre `rect2` within `rect1`. Note that if `rect2` is larger than `rect1` then the final rectangle will lie outside `rect1`.

SEE

`rc_constrain, rc_equal, form_center`

SYNOPSIS

```
#include <aes.h>

res = wind_set(handle, request, x, y, w, h);

int handle;                 window handle
int request;                parameter to set
int x;                      x co-ordinate of rectangle
int y;                      y co-ordinate of rectangle
int w;                      width of rectangle
int h;                      height of rectangle
```

DESCRIPTION

This function sets a particular window attribute. Note that although the binding lists 4 (`int`) parameters only as many as are required need be passed. The actions of the function are defined by the `request` parameter:

Name	Action
WF_NAME	This sets the name or title of the window. Note that due to the 16 bit nature of the binding, the address character pointer passed must be split into it's high and low words. The ADDR macro is provided for this purpose. Alternatively the non-portable `wind_title` function may be used.
WF_INFO	This sets the information line of the window. Like WF_NAME the ADDR macro may be used to perform the word splitting required. Alternatively the non-portable `wind_info` function may be used.
WF_CURRXYWH WF_CXYWH	Set the current position and size of the window including borders. All four parameters are required. Note that if as a result of this call the window size increases in either direction, or if a new part is uncovered then a redraw message will be sent to you by the AES. If you must always redraw as a result of this call then, rather than simply redrawing you should send yourself a redraw message which the AES will merge with any it may have generated automatically.

WF_HSLIDE	x contains the current position of the horizontal slider between 1 and 1000. 1 is the left most position. Note that you should take into account the length of the slider bar when adjusting this value.
WF_VSLIDE	x contains the current position of the vertical slider between 1 and 1000. 1 is the top most position. Note that you should take into account the length of the slider bar when adjusting this value.
WF_TOP	The window specified by handle is the window which you want the AES to place on top (i.e. make the active window).
WF_NEWDESK	This is used to change the object tree for the Desktop to draw. Like WF_NAME the ADDR macro may be used to perform the word splitting required. The first object to draw should be passed as the w parameter. Alternatively the non-portable wind_newdesk function may be used. If you use this call, you should call it again prior to terminating with a (x, y) parameter of NULL to reinstate the default Desktop's tree.
WF_HSLSIZE	x contains the size of the horizontal slider (1 to 1000) or -1 for the default square box.
WF_VSLSIZE	x contains the size of the vertical slider (1 to 1000) or -1 for the default square box.
WF_COLOR	set topped window part indicated by x to colour word y, untopped to w.
WF_DCOLOR	set default topped window part indicated by x to colour word y, untopped to w. Note that applications should *not* use this call, WF_COLOR should be used instead, if required.

The window colour types (WF_COLOR and WF_DCOLOR) are extensions present in TT and MegaST^{E} TOS. The window part whose attribute is to be changed is passed in x, the 'topped' AES colour word in y and the 'untopped' colour word in w. The window parts are as follows:

Symbol	Meaning
W_BOX	Window parent object
W_TITLE	Parent of closer, name and fuller
W_CLOSER	Close box
W_NAME	Mover bar

W_FULLER	Full box
W_INFO	Info line
W_DATA	Holds remainder of window elements
W_WORK	Application work area
W_SIZER	Sizer box
W_VBAR	Parent of vertical slider elements
W_UPARROW	Up arrow
W_DNARROW	Down arrow
W_VSLIDE	Vertical bar
W_VELEV	Vertical elevator
W_HBAR	Parent of horizontal slider elements
W_LFARROW	Left arrow
W_RTARROW	Right arrow
W_HSLIDE	Horizontal bar
W_HELEV	Horizontal elevator

The 'topped' and 'untopped' colour values are standard AES object colour words, i.e.

Border colour	Text colour	Transparent / Opaque	Fill pattern	Fill colour
15 - 12	11 - 8	7	6 - 4	3 - 0

Note that if either parameter has the value -1 then that part is unchanged.

RETURNS

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

SEE

`wind_get, wind_title, wind_info, wind_newdesk`

cookie.h

cookie.h provides facilities for examining and modifying the BIOS cookie jar. The cookie jar is a predefined area of memory which is set aside for system information (e.g. the `_CPU` cookie), or for auto folder/CPX communication (e.g. the `BELL` cookie).

SYNOPSIS

```
#include <cookie.h>

status=getcookie(cookie, pvalue);

int status;              0 if cookie not present
long cookie;             cookie to search for
long *pvalue;            pointer to value found
```

DESCRIPTION

The `getcookie` function searches the BIOS cookie jar attempting to locate the named `cookie`. If the cookie is found then the value is stored in the location pointed to by `pvalue`, if `pvalue` is non-NULL. Note that typical cookie names are 4 character identifiers, so the *Allow multi-character constants* (`-cm`) option must be used if they are to be specified as character constants, e.g. '_CPU'.

The various system cookies, which you may wish to interrogate, are as follows (note that the identifiers `_CPU` etc. are defined in the `cookie.h` header file):

`_CPU`	the bottom 2 digits of the main processor number (e.g. $0 for 68000, $1E for 68030)
`_FDC`	This gives an indication of the highest density floppy unit installed in the machine. The high byte of its value indicates the highest density floppy present: 0 360Kb/720Kb (double-density) 1 1.44Mb (high-density) 2 2.88Mb (extra-high-density) The low three bytes give an indication of the origin of the unit, the value 0x415443 ('ATC') indicates an Atari line-fit or retro-fitted unit.
`_FPU`	This gives an indication of any floating point unit installed in the machine. Only the high word is used at the time of writing. The bits are used as follows (when set): 0 I/O mapped 68881 (e.g. Atari's SFP004) 1 68881/68882 (unsure which) 2 If bit 1 == 0 then 68881, else 68882 3 68040 internal floating point support

_FRB	'Fast RAM Buffer'. This is used on the TT to give the address of a 64K buffer in ST RAM that all ACSI devices performing DMA can use, when transfers to TT RAM are requested. It is not present if there is no fast RAM.
_MCH	This gives the machine type; it consists of a minor number (low word) and a major number (high word) as follows: Major Minor Machine 0 0 520/1040 or Mega ST 1 0 STe 1 16 Mega STe 2 0 TT Normally you should use the more specific cookies given above, in case some one has added a 68030 processor to an STe, for example. One possible use for this cookie is to detect the presence of the extra TT serial ports.
_SND	This is bit oriented as follows: bit 0 1 if ST style GI/Yamaha chip available bit 1 1 if TT/STe style DMA sound available
_SWI	The STe and TT have internal configuration switches; this gives their value.
_VDO	the major/minor part number of the video shifter. At present the least significant word is always zero and the high word is one of: 0 ST 1 STe 2 TT

Note that the absence of a cookie *indicates nothing* about the state of a resource; the host machine may have a 68030, 68882 and high-density floppy fitted with no indication from the cookie jar.

RETURNS

The function returns 0 to indicate that no cookie could be found, or non-zero if the cookie was found. If `pvalue` is non-`NULL` then the value of the cookie is stored in the location pointed to by `pvalue`. The external variable, `__cookie`, is also initialised to the number of the cookie, if found, in the jar.

SEE

putcookie

EXAMPLE

```
#include <cookie.h>
#include <stdio.h>

/* Check what machine we are running on */
void show_machine(void)
{
   long mch=0;

   getcookie(_MCH,&mch);
   switch(mch>>16) {
      case 0:
         puts("520/1040 or Mega ST");
         break;

      case 1:
         switch(mch&0xffff) {
            case 0:
               puts("STE");
               break;

            case 16:
               puts("Mega STE");
               break;

            default:
               puts("Unknown");
               break;
         }
         break;

      case 2:
         puts("TT");
         break;

      default:
         puts("Unknown");
         break;
   }
}
```

SYNOPSIS

```
#include <cookie.h>

status=putcookie(cookie, value);

int status;              0 if could not insert cookie
long cookie;             cookie to insert
long value;              value to give inserted cookie
```

DESCRIPTION

The `putcookie` function places the named cookie into the BIOS cookie jar. If the cookie existed previously it is replaced, otherwise the a new cookie entry is created for the cookie. If no cookie jar is present then one is created and the necessary support code installed (e.g. on pre-TOS 1.06 machines).

When choosing a name for a cookie note that all names starting with an _ in the high byte are reserved for use by Atari. Also, if a program is to use this function it *must* terminate and stay resident (TSR) otherwise the machine may crash badly at a later point.

RETURNS

The function returns 0 to indicate that the cookie could not be installed for some reason (e.g. not enough free memory), or non-zero otherwise.

SEE

`getcookie`

cpx.h

To support the new Atari control panel (XControl) Lattice C 5.5 provides support for generating CPXs, the modules loaded by XControl. These provide an extremely flexible range of user interface options, built into XControl, reducing the size and workload of the individual CPX.

CPXs are designed for controlling a function of the computer, they are *not* an application or desk accessory, although they share many of the restrictions of desk accessories. Writing a CPX is *not* easy, it is strictly a pastime for the experienced programmer who understands both the calling mechanisms used and the call-back processes which occur in such a highly event driven situation.

Because of the restrictive nature of the interface to XControl a CPX *must* use the *Type based stack alignment* (`-aw`) option of the compiler, and must ensure that all functions both have, and are called in the scope of a prototype.

CPX functions

The CPX functions are functions which must be supplied by *your* CPX application. Most of these are optional, and needed only for event CPXs, however *every* CPX must have `cpx_init` (this is the CPX equivalent of a normal program's `main`).

Note that a discussion of *event* and *form* CPXs is beyond the scope of this document and the reader is directed to the disk examples for *form* CPXs, or to Atari's documentation and examples for *event* CPXs.

cpx_button — CPX button event handler

SYNOPSIS

```
#include <cpx.h>

cpx_button(mrets, nclicks, event);

MRETS *mrets;              mouse parameters from event
short nclicks;             number of clicks for event
short *quit;               set non-zero if this event
should terminate the CPX
```

DESCRIPTION

`cpx_button()` is called by XControl when a mouse button event occurs. `mrets` points to an `MRETS` structure which gives details of the mouse event. The definition of `MRETS` is:

```
typedef struct {
   short x;                X co-ordinate of click
   short y;                Y co-ordinate of click
```

```
   short buttons;              button state
   short kstate;               keyboard modifier state
} MRETS;
```

`nclicks` gives the number of clicks which the AES detected. The `*quit` parameter is only used if you wish this event to terminate the CPX; if this is required `*quit` should be set to `1`, otherwise left unmodified.

Note that this function is only required for event CPX's.

RETURNS

None.

EXAMPLE

```
void __stdargs __saveds
cpx_button(MRETS *mrets, short nclicks, short *quit)
{
   extern OBJECT tree[];
   int obj;

   obj = objc_find(tree,ROOT,MAX_DEPTH,mrets->x,mrets->y);
   switch(obj)
      ...
}
```

cpx_call — *CPX interaction handler*

SYNOPSIS

```
   #include <cpx.h>

   flag = cpx_call(rect);

   short flag;                 zero if CPX has finished
   GRECT *rect;                XControl rectangle
```

DESCRIPTION

`cpx_call()` is called by XControl after `cpx_init()` returns. It is used to set up the work area of the CPX. Optionally the CPX may then call `Xform_do()` to manage the user interface.

Note that this function is required for *both* 'form' *and* 'event' CPX's.

RETURNS

cpx_call() should return 0 if it has finished processing events (in the case of an Xform_do() based CPX), or non-zero to indicate that XControl should continue to dispatch events via the CPXINFO hooks.

EXAMPLE

```
int __stdargs __saveds
cpx_call(GRECT *rect)
{
   short button;
   int quit = 0;

   // Try to find the cookie describing the configuration
   if (!xcpb->getcookie(MY_COOKIE,(long *)&cookie)) {
      form_alert(1,cookie_missing);
      return 0;
   }
   // Initialise location of form within CPX window
   tree[ROOT].ob_x = rect->g_x;
   tree[ROOT].ob_y = rect->g_y;

   objc_draw(tree, ROOT, MAX_DEPTH, rect->g_x, rect->g_y,
     rect->g_w, rect->g_h);

   return 1;
}
```

cpx_close — CPX termination handler

SYNOPSIS

```
#include <cpx.h>

cpx_close(flag);

short flag;                 non-zero if WM_CLOSE message
```

DESCRIPTION

cpx_close() is called by XControl whenever a WM_CLOSE or AC_CLOSE message is received. On return from this function the CPX must ensure that no outstanding memory is Malloc()'d.

We strongly recommend that CPX's *never* allocate any memory via `Malloc()`. Note that you should treat `WM_CLOSE` as an OK action, whilst `AC_CLOSE` should be treated as a Cancel action.

`flag` is a parameter indicating whether the message was an `AC_CLOSE` or a `WM_CLOSE`, it is non-zero to indicate the latter.

Note that this function is only required for event CPX's.

RETURNS

None.

EXAMPLE

```
void __stdargs __saveds
cpx_close(short flag)
{
   if (flag)     /* non-zero indicates an OK type event */
      update_settings();
}
```

cpx_draw — CPX redraw event handler

SYNOPSIS

```
#include <cpx.h>

cpx_draw(clip);

GRECT *clip;              dirtied rectangle to redraw
```

DESCRIPTION

`cpx_draw()` is called by XControl whenever a `WM_REDRAW` message is received so that the CPX may redraw the dirtied rectangle given by `clip`. In order to do this correctly it should 'walk' the rectangle list using the XControl utility functions `GetFirstRect()` and `GetNextRect()`.

Note that this function is only required for event CPX's.

RETURNS

None.

EXAMPLE

```
void __saveds __stdargs
cpx_draw(GRECT *clip)
{
   extern OBJECT tree[];

   clip = xcpb->GetFirstRect(clip);
   while(clip)
   {
      objc_draw(tree, ROOT, MAX_DEPTH,
        clip->g_x, clip->g_y, clip->g_w, clip->g_h);
      clip = xcpb->GetNextRect();
   }
}
```

cpx_hook — CPX event pre-emption handler

SYNOPSIS

```
   #include <cpx.h>

   override = cpx_hook(event, msg, mrets, key, nclicks);

   short override;          non-zero to inhibit default
                            event handling
   short event;             event mask
   short *msg;              AES event message buffer
   MRETS *mrets;            mouse parameters
   short *key;              key returned
   short *nclicks;          number of button clicks
```

DESCRIPTION

`cpx_hook()` is called by XControl immediately after receipt of an event from `evnt_multi()`. It may be used to dispatch messages in a manner not normally available from XControl.

`event` gives the event mask supplied to `evnt_multi()`. `msg` is a pointer to the received message packet. `mrets` points to an `MRETS` structure which gives details of the mouse event. The definition of `MRETS` is:

```
typedef struct {
   short x;                 X co-ordinate of click
   short y;                 Y co-ordinate of click
```

```
   short buttons;              button state
   short kstate;               keyboard modifier state
} MRETS;
```

*nclicks gives the number of clicks which the AES detected; note that this parameter may be modified if required. *key gives the key detected (if any); again this parameter may be modified if required.

Note that this function is only required for event CPX's.

RETURNS

cpx_hook() should return 0 to continue with the default CPX event handling, or non-zero to inhibit it.

EXAMPLE

```
short cpx_hook(short event, short *msg, MRETS *mrets,
  short *key, short *nclicks)
{
   return 0;
}
```

cpx_init — *Main CPX entry point*

SYNOPSIS

```
   #include <cpx.h>

   CPXINFO *cpx_init(xcpb);   main CPX entry point

   XCPB *xcpb;                XControl parameter block
```

DESCRIPTION

cpx_init() is the main CPX entry point. The routine is called at boot time and also whenever the user invokes the CPX. A pointer to the XControl parameter block is passed in xcpb to the routine. The parameter block contains the following information:

```
typedef struct {
   short handle;              AES' VDI workstation handle
   short booting;             non-zero if booting
   short version;             XControl version number
   short SkipRshFix;          zero if resource should be fixed up
```

```
    char *reserve1;             reserved
    char *reserve2;             reserved
    void (*rsh_fix)(…);         resource file fixup routine
    void (*rsh_obfix)(…);       OBJECT fixup routine
    short (*Popup)(…);          popup menu handler
    void (*Sl_size)(…);         size slider routine
    void (*Sl_x)(…);            X-position slider routine
    void (*Sl_y)(…);            Y-position slider routine
    void (*Sl_arrow)(…);        arrow slider handler
    void (*Sl_dragx)(…);        X-drag slider handler
    void (*Sl_dragy)(…);        Y-drag slider handler
    short (*Xform_do)(…);       XControl form_do()
    GRECT *(*GetFirstRect)(…); get first redraw rectangle
    GRECT *(*GetNextRext)(…);   get next redraw rectangle
    void (*Set_Evnt_Mask)(…);   set XControl event mask
    short (*XGen_Alert)(…);     generate XControl alert
    short (*CPX_Save)(…);       save CPX configuration
    void *(*Get_Buffer)(…);     get pointer to CPX buffer
    short (*getcookie)(…);      get value from cookie jar
    short Country_Code;         country code of XControl
    void (*MFsave)(…);          save mouse form
} XCPB;
```

Most of the fields in this structure are pointers to XControl utility functions, which are all described below. The remaining fields are used as follows:

`handle`	The physical workstation handle obtained via `graf_handle()`.
`booting`	Non-zero if this call to `cpx_init()` is part of the XControl initialisation sequence. Note that a CPX must have the `CPX_BOOTINIT` or `CPX_SETONLY` flag set in the CPX header for this call to be made.
`version`	The version number of XControl which is active. At the time of writing the version number is 0.
`SkipRshFix`	Zero if resource should be fixed up. Note that it is up to the user to make this flag non-zero after any one-shot initialisation has been performed.
`Country_Code`	Country code of country for which XControl was compiled. Note that the values used are identical to those in the external variable `_country`.

RETURNS

`cpx_init()` must return a pointer to a structure of type `CPXINFO`. This has the definition:

```
typedef struct {
   short (*cpx_call)(…);      CPX invocation routine
   void (*cpx_draw)(…);       CPX redraw routine
   void (*cpx_wmove)(…);      CPX window moved routine
   void (*cpx_timer)(…);      CPX timer event routine
   void (*cpx_key)(…);        CPX keyboard event routine
   void (*cpx_button)(…);     CPX button event routine
   void (*cpx_m1)(…);         CPX mouse rectangle event 1
   void (*cpx_m2)(…);         CPX mouse rectangle event 2
   short (*cpx_hook)(…);      CPX event pre-emption hook
   void (*cpx_close)(…);      CPX termination routine
} CPXINFO;
```

With the exception of the `cpx_call()` these fields are only used by 'event' CPXs and should be `NULL` for 'form' CPXs.

If the call to `cpx_init()` was made as part of the XControl initialisation sequence (i.e. the `xcpb->booting` flag is non-zero) then the CPX should return the value `NULL` if no further events should be dispatched via it (i.e. XControl may relinquish the CPX header memory), or the value `(CPXINFO *)1` if further events are required.

Otherwise the CPX should return a pointer to a `static CPXINFO` structure containing pointers to the relevant handlers. Note that *under no circumstances* may an automatic structure be used for this purpose.

The details of the handler functions are contained elsewhere in this section.

EXAMPLE

```
#include <cpx.h>
#include <acc.h>

XCPB *xcpb;                /* XControl Parameter Block */

CPXINFO * __stdargs __saveds
cpx_init(XCPB * Xcpb)
{
   static long heap[16384/sizeof(long)];   /*heap space */

   xcpb = Xcpb;

   if (xcpb->booting) {
      /*
       * Try to find our cookie
       */
      if (xcpb->getcookie(MY_COOKIE,(long *)&cookie)) {
         …
      }
      return (CPXINFO *) 1;   /* to keep going */
   }
   else {
      static CPXINFO cpxinfo = {cpx_call};

      appl_init();    /* initialise private tables */

      if(!xcpb->SkipRshFix) {
         xcpb->SkipRshFix=1;
         _addheap(heap,sizeof(heap)); /* only once */
      }
      return &cpxinfo;
   }
}
```

SYNOPSIS

```
#include <cpx.h>

cpx_key(kstate, key, event);

short kstate;              state of modifiers (Ctrl, Alt
etc.)
short key;                 key pressed
short *quit;               set non-zero if this event
                           should terminate the CPX
```

DESCRIPTION

`cpx_key()` is called by XControl when a keyboard button event occurs. `key` gives the key pressed; the bottom eight bits are the ASCII code for the character. The top eight bits are the scan code for the key. The `kstate` parameter gives the state of the shift keys depressed; this is a bitmap with the following meanings:

Name	Value	Meaning
K_RSHIFT	0x0001	Right shift key depressed
K_LSHIFT	0x0002	Left shift key depressed
K_CTRL	0x0004	Ctrl key depressed
K_ALT	0x0008	Alt key depressed

The `*quit` parameter is only used if you wish this event to terminate the CPX; if this is required `*quit` should be set to `1`, otherwise left unmodified.

Note that this function is only required for event CPX's.

RETURNS

None.

EXAMPLE

```
void __stdargs __saveds
cpx_key()
{
   extern OBJECT tree[];
   int obj;
```

```
   obj = objc_find(tree,ROOT,MAX_DEPTH,mrets->x,mrets->y);
   switch(obj)
      ...
}
```

cpx_m1, cpx_m2 CPX mouse rectangle event handler

SYNOPSIS

```
   #include <cpx.h>

   cpx_m1(mrets, quit);
   cpx_m2(mrets, quit);

   MRETS *mrets;              mouse parameters from event.
   short *quit;               set non-zero if this event
                              should terminate the CPX
```

DESCRIPTION

`cpx_m1()` and `cpx_m2()` are called by XControl when a mouse rectangle event occurs. `mrets` points to an `MRETS` structure which gives details of the mouse event. The definition of `MRETS` is:

```
typedef struct {
   short x;   X co-ordinate of click
   short y;   Y co-ordinate of click
   short buttons;  button state
   short kstate;   keyboard modifier state
} MRETS;
```

The `*quit` parameter is only used if you wish this event to terminate the CPX; if this is required `*quit` should be set to `1`, otherwise left unmodified.

Note that this function is only required for event CPX's.

RETURNS

None.

EXAMPLE

cpx_timer — CPX timer event handler

SYNOPSIS

```
#include <cpx.h>

cpx_timer(quit);

short *quit;              set non-zero if this event
                          should terminate the CPX
```

DESCRIPTION

`cpx_timer()` is called by XControl when a timer event occurs. The `*quit` parameter is only used if you wish this event to terminate the CPX; if this is required `*quit` should be set to `1`, otherwise left unmodified.

Note that this function is only required for event CPX's, also note that 'form' CPXs cannot handle timer events. If such events are necessary then you should design your CPX as an 'event' CPX.

RETURNS

None.

EXAMPLE

cpx_wmove — CPX window move event handler

SYNOPSIS

```
#include <cpx.h>

cpx_wmove(rect);

GRECT *rect;              XControl rectangle
```

DESCRIPTION

`cpx_wmove()` is called by XControl on receipt of a window moved message. `rect` contains the new window position and size.

RETURNS

None.

EXAMPLE

```
void __saveds __stdargs
cpx_wmove(GRECT *rect)
{
   extern OBJECT tree[];

   tree[ROOT].ob_x = rect->g_x;
   tree[ROOT].ob_y = rect->g_y;
}
```

XControl utility functions

XControl provides a number of utility functions for use by a CPX; these functions are all accessed using indirect function calls. The address of the functions are contained in a parameter block passed to the `cpx_init()` function.

Within the following discussion, the mechanics of the calling are ignored, although this is almost always of the form:

```
xpcb->Xform_do(…);
```

where `xpcb` is the pointer passed to `cpx_init()`.

SYNOPSIS

```
#include <cpx.h>

err = CPX_Save(void *ptr, num);

int err;                    zero if error occurred
void *ptr;                  data to be saved.
long num;                   number of bytes to write
```

DESCRIPTION

CPX_Save() is used to write any configuration information, which the user has requested be saved, back to the CPX file. `ptr` is used to point to the area of memory which holds the configuration information block, whilst `num` gives the number of bytes to be written.

This information is always saved directly into the .CPX file at the start of the data section, overwriting whatever is already there. As such the executable must be carefully constructed to ensure that this is the case. This can be done by ensuring that the very first file which is linked contains the initialised configuration block in the far data section (as shown below). Note that on subsequent reloads the contents of this block will reflect the last saved values.

RETURNS

The return value is non-zero on success, otherwise 0 to indicate a failure.

EXAMPLE

```
#include <cpx.h>

XPCB *xpcb;

struct {
   int config1, config2, config3;
} __far configuration = {
   0, 0, 0;   /* note this must be initialised */
};
...
   xpcb->CPX_Save(&configuration, sizeof(configuration));
```

Get_Buffer — Get pointer to static CPX buffer

SYNOPSIS

```
#include <cpx.h>

ptr = Get_Buffer();

void *ptr;                  pointer to 64 byte buffer
```

DESCRIPTION

`Get_Buffer()` returns a pointer to the 64 byte buffer in the CPX header, available for use by the CPX as static data. Because a CPX is normally reloaded on each execution, settings are normally forgotten between executions. By careful use of this buffer this need not occur.

RETURNS

The function returns a pointer to the 64 byte static buffer.

getcookie — Locate cookie jar entry

SYNOPSIS

```
#include <cpx.h>

status=getcookie(cookie, pvalue);

short status;               0 if cookie not present
long cookie;                cookie to search for
long *pvalue;               pointer to value found
```

DESCRIPTION

The `getcookie` function searches the BIOS cookie jar attempting to locate the named `cookie`. If the cookie is found then the value is stored in the location pointed to by pvalue, if pvalue is non-NULL. Note that typical cookie names are 4 character identifiers, so the *Allow multi-character constants* (`-cm`) option must be used if they are to be specified as character constants, e.g. '`_CPU`'.

You should use this routine rather than the runtime library version when looking for a cookie from a CPX. Typically a cookie may be used by an AUTO folder TSR to indicate where the CPX may find the configuration data used by the TSR.

RETURNS

The function returns 0 to indicate that no cookie could be found, or non-zero if the cookie was found. If `pvalue` is non-`NULL` then the value of the cookie is stored in the location pointed to by `pvalue`.

GetFirstRect, GetNextRext Obtain XControl rectangle

SYNOPSIS

```
#include <cpx.h>

rdrw = GetFirstRect(rect);
rdrw = GetNextRext();

GRECT *rdrw;              intersecting GRECT for redraw
GRECT *rect;              dirtied rectangle
```

DESCRIPTION

When an event CPX must redraw due a to `WM_REDRAW` message the CPX must obtain the rectangle list via these operations.

RETURNS

A pointer to a `GRECT` to redraw or NULL.

SYNOPSIS

```
#include <cpx.h>

const int MFSAVE, MFRESTORE;

MFsave(saveit,mf);

short saveit;              MFSAVE or MFRESTORE
MFORM *mf;                 mouse form buffer
```

DESCRIPTION

`MFsave()` is used to save/restore a mouse pointer. If for example a CPX wishes to switch to a flat-hand pointer whilst dragging, it must restore the pointer to the application shape after the call.

`saveit` is a flag indicating whether the mouse pointer is to be saved or restored and has the values `MFSAVE` or `MFRESTORE`. `mf` is a pointer to an `MFORM` structure in which the mouse pointer is saved to/restored from.

RETURNS

None.

EXAMPLE

```
#include <cpx.h>

XPCB *xpcb;

MFORM mf;
...
xcpb->MFsave(MFSAVE, &mf);
graf_mouse(BUSY_BEE, NULL);
...
xcpb->MFsave(MFRESTORE, &mf);
```

SYNOPSIS

```
#include <cpx.h>

select = Popup(items, num_items, default_item,
                          font_size,button, world);

short select;             item selected, or -1
const char *items[];      pointer to array of strings
short num_items;          number of items
short default_item;       the default item, or -1
short font_size;          3
const GRECT *button;      GRECT of button used to
                          invoke popup.
const GRECT *world;       GRECT of bounding box
```

DESCRIPTION

The `Popup()` call is used to display and interact with popup menus. A pointer to an array of strings is passed, giving the names of the elements, in `items`. Each string should be padded at the start with two spaces, and then the lengths padded to the length of the longest string plus 1, with spaces.

The number of elements in the array is passed in `num_items`. `default_item` gives the current selection, this item will be marked with a check mark; note that if you require no default item, pass the value -1.

The `font_size` variable should always indicate the large font be used, this is equivalent to the constant `IBM` in `aes.h`. `button` is used to indicate the rectangle of the button which caused the popup to appear. This is used to ensure that the menu which pops-up is correctly centred on the original item.

`world` gives the bounding box within which the popup is to appear. This ensures that it cannot, for example, popup outside the main CPX window. Typically this box will be the size of your entire CPX form (i.e. 256 * 176 pixels).

RETURNS

The function returns the number of the string which the user selected or -1 if the operation was cancelled (clicked off the popup).

EXAMPLE

```
static char *items[] = {
   poptext_1,
   poptext_2,
   poptext_3,
   poptext_4,
};

GRECT clip, world;
short curitem, obj;
...
switch (button) {
   case popup:
      /*
       * Obtain rectangle of popup activation button
       * and call the popup draw/handle routine.
       */
      objc_xywh(tree, button, &clip);
      obj = xcpb->Popup(items,
        sizeof(items)/sizeof(items[0]), curitem,
        IBM, &clip, &world);

      /*
       * If an object was actually selected, then update
       * our settings.
       */
      if (obj!=NIL)
         curitem=obj;

      /*
       * Redraw the popup button.
       */
      objc_draw(tree, ROOT, MAX_DEPTH, ELTS(clip));
      break;
...
}
```

SYNOPSIS

```
#include <cpx.h>

rsh_fix(num_objs, num_frstr, num_frimg, num_tree,
                          rs_object, rs_tedinfo,
                          rs_strings, rs_iconblk,
                          rs_bitblk, rs_frstr,
                          rs_frimg, rs_trindex,
                          rs_imdope );

int num_objs;             number of objects
int num_frstr;            number of free strings in
int num_frimg;            number of free images
int num_tree;             number of trees
OBJECT *rs_object;        pointer to first OBJECT
TEDINFO *rs_tedinfo;      pointer to first TEDINFO
char *rs_strings[];       pointer to string table
ICONBLK *rs_iconblk;      pointer to first ICONBLK
BITBLK *rs_bitblk;        pointer to first BITBLK
long *rs_frstr;           pointer to free strings
long *rs_frimg;           pointer to free images
long *rs_trindex;         pointer to tree index
struct foobar *rs_imdope; pointer to image structures
```

DESCRIPTION

`rsh_fix()` is used to fix up an RCS 2 style embedded resource file. All of the parameters are designed to be passed directly from the .RSH file created by RCS 2. Because CPX resources are always based on the 8x16 pixel font, normally DERCS is used to do this fixup at compile time.

Note that if you are using this function you must ensure that it is done only once. This is achieved using the `SkipRshFix` flag, which on the first call to the CPX will be zero. Note that you must set this to 1 manually after performing any initialisation.

RETURNS

None.

rsh_obfix — *Fix-up single object*

SYNOPSIS

```
#include <cpx.h>

rsh_obfix(tree, curob);

OBJECT *tree;              the object tree to convert
int curob;                 the object number to convert
```

DESCRIPTION

The `rsh_obfix()` is used to fix up a single object within a tree in a similar manner to `rsrc_obfix()`. This function is intended for use in fixing up resources which are not suitable for `rsh_fix()`. When using WERCS/DERCS none of these functions are required.

RETURNS

None.

Set_Evnt_Mask — *Set event CPX message mask*

SYNOPSIS

```
#include <cpx.h>

Set_Evnt_Mask(mask, m1, m2, time);

short mask;                events to receive
MOBLK *m1;                 mouse rectangle 1
MOBLK *m2;                 mouse rectangle 2
long time;                 time delay to wait for
```

DESCRIPTION

`Set_Evnt_Mask` is used to initialise the event mask for *event* CPXs. `mask` gives the required mask (`MU_KEYBD | MU_BUTTON |` ... etc.), `m1` and `m2` give the mouse rectangles, whilst `time` gives the `evnt_multi` delay parameter in milliseconds.

SYNOPSIS

```
#include <cpx.h>

Sl_arrow(tree, base, slider, arrow, change, min, max,
                          pvalue, direction, rdrw);

OBJECT *tree;             pointer to tree
short base;               the base of the slider
short slider;             the elevator of the slider
short arrow;              the arrow clicked on
short change;             requested change value
short min;                minimum value possible
short max;                maximum value possible
short *pvalue;            pointer to current value
short direction;          VERTICAL or HORIZONTAL
void (*rdrw)(void);       pointer to redraw function,
                          or NULL
```

DESCRIPTION

`Sl_arrow()` is used to control the action of the arrows on a slider bar when the user clicks on an arrow. `tree` is a pointer to the resource tree containing the slider control, `base` gives the object number of the base of the slider object (within which the slider operates), `slider` gives the object number of the elevator (the object which does the sliding). `arrow` is the object number which is to be highlighted during this operation, or `NIL` to indicate no highlighting is required.

`change` gives the amount by which the value should change for each 'click' on the arrow (±1). `direction` gives the direction in which the slider is to operate. This should have the value `VERTICAL` or `HORIZONTAL` as defined in aes.h.

`pvalue` is a pointer to the current setting of the control, this value will be updated as a result of this call. `min` and `max` give the maximum and minimum values for pvalue. For a `VERTICAL` slider the minimum value is towards the bottom of the tree, whilst for a `HORIZONTAL` one it is towards the left of the tree; if you need the sliders to operate in an inverse manner, simply swap the `max` and `min` values in the `Sl_arrow()` call.

In order to make the slider 'active', i.e. have the information presented updated during the operation, `rdrw` should point to a function which redraws the window contents based on the setting of `*pvalue`. If you do not require continuous updating, simply pass the value `NULL`.

This function may also be used to implement paging (i.e. when the user clicks on the base of the slider). This is effected by passing a value larger than ±1 as the change factor.

RETURNS

None.

EXAMPLE

```
MRETS mk;
short ox, oy, value = 0;
…
switch (button) {
   case uparrow:
   case dnarrow:
      /*
       * User is manipulating one of the arrow keys, so
       * call the XControl arrow handling code.
       */
      xcpb->Sl_arrow(tree, base, slider, button,
        button == uparrow?-1:1,
        max, 0, &value, VERTICAL, redraw);
      break;

   case base:
      /*
       * This is a click on the bar behind the slider,
       * i.e. a page up or page down request.
       */
      graf_mkstate(&mk.x, &mk.y, &mk.buttons, &mk.kstate);
      objc_offset(bell_box, bell_slider, &ox, &oy);

      /*
       * Decide whether it was up or down and move by the
       * number of lines in a window less 1, this ensures
       * that you can always see what was at the limit of
       * the window before the action.
       */
      ox = (mk.y < oy) ? -(NLINES-1) : (NLINES-1);
```

```
        xcpb->Sl_arrow(tree, base, slider, -1, ox,
          max, 0, &value, VERTICAL, redraw);
        break;
    ...
}
```

Sl_dragx, Sl_dragy — *Slider dragging control*

SYNOPSIS

```
#include <cpx.h>

Sl_dragx(tree, base, slider, min, max, pvalue, rdrw);
Sl_dragy(tree, base, slider, min, max, pvalue, rdrw);

OBJECT *tree;              pointer to tree
short base;                the base of the slider
short slider;              the elevator of the slider
short min;                 minimum value possible
short max;                 maximum value possible
short *pvalue;             pointer to current value
void (*rdrw)(void);        pointer to redraw function,
                           or NULL
```

DESCRIPTION

`Sl_dragx()` and `Sl_dragy()` are used to control the action of the user dragging the `elevator` of a slider bar control.

`tree` is pointer to the resource tree containing the slider control, `base` gives the object number of the base of the slider object (within which the slider operates), `slider` gives the object number of the elevator (the object which does the sliding).

`pvalue` is a pointer to the current setting of the control, this value will be updated as a result of this call. `min` and `max` give the maximum and minimum values for `pvalue`. For a `VERTICAL` slider the minimum value is towards the bottom of the tree, whilst for a `HORIZONTAL` one it is towards the left of the tree; if you need the sliders to operate in an inverse manner, simply swap the `max` and `min` values in the `Sl_dragx()/Sl_dragy()` call.

In order to make the slider 'active', i.e. have the information presented updated during the operation, `rdrw` should point to a function which redraws the window contents based on the setting of `*pvalue`. If you do not require continuous updating, simply pass the value `NULL`.

RETURNS

None.

EXAMPLE

```
short value = 0;
MFORM Mbuffer;
...
switch (button) {
   case slider:
      /*
       * Slider is being dragged, again just call the
       * XControl routine to do most of the work.
       */
      xcpb->MFsave(MFSAVE, &Mbuffer);
      graf_mouse(FLAT_HAND, NULL);
      xcpb->Sl_dragy(tree, base, slider, max, 0, &value,
        redraw);
      xcpb->MFsave(MFRESTORE, &Mbuffer);
      break;
...
}
```

SYNOPSIS

```
#include <cpx.h>

Sl_size(tree, base, slider, range, visible, direction,
                          min_size);

OBJECT *tree;             pointer to tree
short base;               the base of the slider
short slider;             the elevator of the slider
short range;              total number of items
short visible;            number of visible items
short direction;          VERTICAL or HORIZONTAL
short min_size;           minimum pixel size of slider
```

DESCRIPTION

`Sl_size()` is used to adjust the size of a slider within the base so that it is proportional to the amount of data present. `tree` is pointer to the resource tree containing the slider control, `base` gives the object number of the base of the slider object (within which the slider operates), `slider` gives the object number of the elevator (the object which does the sliding). `direction` gives the direction in which the slider is to operate. This should have the value `VERTICAL` or `HORIZONTAL` as defined in aes.h.

`range` is the total number of items which are available, whilst `visible` gives the number of such items which are visible at any one time. `min_size` is used to fix a minimum pixel height/width for the slider, typically this will have the value 1 or 2. A situation in which a larger size may be desirable is if the slider has some text embedded in it.

RETURNS

None.

EXAMPLE

```
#include <aes.h>
#include <cpx.h>

short range;
…
xcpb->Sl_size(tree, base, slider, max + NLINES, NLINES,
  VERTICAL, 2);
```

Sl_x, Sl_y — Update slider position

SYNOPSIS

```
#include <cpx.h>

Sl_x(tree, base, slider, value, min, max, rdrw);
Sl_y(tree, base, slider, value, min, max, rdrw);

OBJECT *tree;            pointer to tree
short base;              the base of the slider
short slider;            the elevator of the slider
short value;             new value
short min;               minimum value possible
short max;               maximum value possible
void (*rdrw)(void);      pointer to redraw function,
                         or NULL
```

DESCRIPTION

`Sl_x()` and `Sl_y()` are used to reposition the elevator of a slider control at the program's request. Typically these calls are used when the user is not manipulating the slider control directly, but some related action requires that the slider be updated.

`tree` is pointer to the resource tree containing the slider control, `base` gives the object number of the base of the slider object (within which the slider operates), `slider` gives the object number of the elevator (the object which does the sliding).

`value` is the new setting required for the control; note that this is *not* a pointer as in other CPX slider calls. `min` and `max` give the maximum and minimum values for `pvalue`. For a `VERTICAL` slider the minimum value is towards the bottom of the tree, whilst for a `HORIZONTAL` one it is towards the left of the tree; if you need the sliders to operate in an inverse manner, simply swap the `max` and `min` values in the `Sl_x()/Sl_y()` call.

In order to make the slider 'active', i.e. the information presented is updated during the operation, `rdrw` should point to a function which redraws the window contents based on the setting of `*pvalue`. If you do not require continuous updating, simply pass the value `NULL`.

Note that this does not update the on screen slider, this must be accomplished manually (if required) by an `objc_draw()` call.

RETURNS(*)

None.

EXAMPLE

```
short value = 0;
GRECT clip1, clip2;
...
value--;
objc_xywh(tree, slider, &clip1);
xcpb->Sl_y(tree, base, slider, value, max, 0, redraw);
objc_xywh(tree, slider, &clip2);
rc_union(&clip2, &clip1);
objc_draw(tree, ROOT, MAX_DEPTH,
  clip1.g_x, clip1.g_y, clip1.g_w, clip1.g_h);
```

SYNOPSIS

```
#include <cpx.h>

res = Xform_do(tree, startob, msg);

short res                   exit object index
OBJECT *tree;               object tree of the form
short startob;              editable object to start with
short *msg;                 message buffer
```

DESCRIPTION

This function is used to let the user fill in a form or dialog box displayed within the XControl window. The `tree` parameter is the address of the form; note that the size of this form should be *exactly* 256*176 pixels. XControl needs to know which editable text item to display the initial text cursor; this is passed in the `startob` parameter. If there are no editable text fields, or you wish to start editing at the first editable field then 0 should be used.

`msg` is a pointer to an 8 entry `short` array which is used to hand-off messages which XControl wishes the CPX to handle. The messages generated are:

msg[0]	Action
WM_REDRAW	Part of the XControl window needs redrawing in a manner which it cannot handle.In order to do this correctly it should 'walk' the rectangle list using the XControl utility functions `GetFirstRect()` and `GetNextRect()`.
AC_CLOSE WM_CLOSE	The window is being closed, either explicitly (WM_CLOSE) or implicitly (AC_CLOSE). At this time the CPX must be certain that no outstanding memory is `Malloc()`'d. We strongly recommend that CPX's *never* allocate any memory via `Malloc()`. Note that you should treat WM_CLOSE as an OK action, whilst AC_CLOSE should be treated as a Cancel action.
CT_KEY	A key was pressed; `msg[3]` contains the keycode of the key. Note that only non-printing keys are returned (F1-F10 etc.), but since other keystrokes are used by the editable text handler cursor keys, Tab etc. can *never* be returned.

RETURNS

This function returns the object index of the item that caused the dialog to finish (e.g. that of an OK button). Your program can then compare this with the values in the resource header file. Note that the value returned may be negative indicating that the exit object was double clicked in which case the bottom 15 bits should be masked off to find the true exit button. Also the exit object is not automatically de-selected when `Xform_do` returns, so you should do this manually.

If the special value -1 is returned this indicates the the CPX should examine the `msg` array for a message which XControl wishes the CPX to handle.

EXAMPLE

```
int __stdargs __saveds
cpx_call(GRECT *rect)
{
   short button;
   int quit=0;
…
   do {
      int double_click;
      short msg[8];

      /*
       * Waiting for a reply
       */
      button = xcpb->Xform_do(bell_box, 0, msg);

      /* Check if we have a double click item */
      if ((button != -1) && (button & 0x8000)) {
         double_click = 1;
         button &= 0x7FFF;
      }
      else
         double_click=0;
      /*
       * If it wasn't a button then try to turn it into
       * one.
       */
      if (button == -1) {
         switch (msg[0]) {
           case AC_CLOSE:     /* ac_close means cancel */
```

```
            button=bell_btcancel;
            break;

         case WM_CLOSED:    /* wm_close means ok */
            button=bell_btok;
            break;
      }
   }

   switch (button) {
      ...
   }
} while (!quit);

return 0;
}
```

XGen_Alert — *Generate CPX error*

SYNOPSIS

```
#include <cpx.h>

const int FILE_ERR;
const int FILE_NOT_FOUND;
const int MEM_ERR;
const int SAVE_DEFAULTS;

ok = XGen_Alert(id);

short ok;                  zero if user cancelled, else
non-zero
short id;                  id of XControl alert
```

DESCRIPTION

`XGen_Alert` is used to generate an alert centred within the XControl window. The number of the alert required is passed in `id`.

id	Meaning
SAVE_DEFAULTS	Save Defaults?
MEM_ERR	Memory allocation problem
FILE_ERR	File I/O error
FILE_NOT_FOUND	File not found

RETURNS

A zero value is returned to indicate the user selected cancel, or non-zero otherwise. Note that alerts with only one button always return a non-zero value.

EXAMPLE

```
#include <oserr.h>
#include <cpx.h>

void process_err(int fd)
{
   if (fd==-EFILNF)
      xcpb->XGen_Alert(FILE_NOT_FOUND);
   else if (fd<0)
      xcpb->XGen_Alert(FILE_ERR);
}
```

ext.h

ext.h provides compatibility macros and functions for portability to other Atari C compilers. It is not recommended that any of these functions are used in your own programs, instead they should only be used if required when porting software.

coreleft — Estimate remaining memory

SYNOPSIS

```
#include <ext.h>

max = coreleft();

size_t max;                maximum block in system heap
```

DESCRIPTION

This function returns the size of the largest block available in the system heap. Note that this is not the same as the maximum memory available, there may be some additional non-allocated blocks available.

RETURNS

As noted above

SEE

`Malloc`

delay, sleep — Wait for time to elapse

SYNOPSIS

```
#include <ext.h>

delay(ms);              pause for ms milliseconds
sleep(s);               pause for s seconds

size_t ms;              milliseconds to wait
size_t s;               seconds to wait
```

DESCRIPTION

These functions wait for a specified period of time to elapse. `delay` is passed a parameter, `ms`, giving the number of milliseconds to wait for, whilst `sleep` is given the number of seconds, `s`.

Note that unlike similar calls under UNIX®, these calls do not suspend the process, instead they 'busy-wait'.

RETURNS

None

SEE

`clock, difftime`

SYNOPSIS

```
#include <ext.h>

err = findfirst(name,info,attr);      Find first
directory entry
err = findnext(info);      Find next directory entry

int err;                   0 if successful
struct ffblk *info;        file information area
const char *name;          file name or pattern
int attr;                  file attribute bits
```

DESCRIPTION

These functions search a directory for entries that match the specified file name or file name pattern. The `findfirst` function locates the first matching file. Then successive calls to `findnext` locate additional matching files. Each `findnext` call must be given the file information that was returned on the preceding call to `findfirst` or `findnext`.

The `name` argument must be a null-terminated string specifying the drive, path, and name of the desired file. The drive and path can be omitted, in which case the current directory will be searched. You can use the GEMDOS * and ? characters for pattern matching in the name portion. For example, `xy*.b` will locate files in the current directory that begin with `xy` and have `b` as their extension.

The `attr` argument specifies which file types are to be included in the search. The following bits are used:

Bit	Meaning
FA_RDONLY	Read-only flag
FA_HIDDEN	Hidden file flag
FA_SYSTEM	System file flag
FA_LABEL	Volume label flag
FA_DIREC	Subdirectory flag
FA_ARCH	Archive flag

The `info` argument points to a file information structure as defined in the `ext.h` header file. For GEMDOS, this is the same as the GEMDOS DTA structure:

```
struct ffblk {
   char ff_reserved[21];      reserved
   char ff_attrib;            actual file attribute
   short ff_ftime             file time
   short ff_fdate;            file date
   long ff_fsize;             file size in bytes
   char ff_name[14];          file name
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in `errno` and `_OSERR`.

SEE

`Fsfirst, Fsnext, getfnl, errno, _OSERR`

ftimtotm — Convert time structures

SYNOPSIS

```
#include <ext.h>

tm = ftimtotm(ft);          convert time structures

struct tm *tm;              pointer to struct tm
struct ftime *ft;           pointer to struct ftime
```

DESCRIPTION

`ftimtotm` converts a file time structure (`struct ftime`) to the ANSI time `struct tm`. Note that the pointer returned is that shared by `gmtime` and `localtime`, a call to either one will destroy the results of the previous call.

RETURNS

A pointer to a structure of type `struct tm` is returned containing the converted time.

SEE

ftunpk, utpack, gmtime

getcurdir — Get current directory

SYNOPSIS

```
#include <ext.h>

error = getcurdir(drive,path);

int error;              0 if successful
int drive;              drive code
char *path;             points to path area
```

DESCRIPTION

This function gets the current directory path for the specified disk drive. The drive codes are 0 for the current drive, 1 for drive A, 2 for drive B, and so on.

Note that the path area must be large enough to contain the expected path (FMSIZE is a safe value). The returned string will contain the entire path, including the drive name of the device.

RETURNS

If the operation is successful, the function returns 0. Otherwise it returns -1 and places error information in errno and _OSERR.

SEE

Dgetpath, getcd, getcwd, errno, _OSERR

SYNOPSIS

```
#include <ext.h>

getdate(curdate);          get current system date
setdate(newdate);          set current system date

struct date *curdate;      pointer to current date
struct date *newdate;      pointer to date to be set
```

DESCRIPTION

These functions manipulate the system date. `getdate` fills in the structure of type `date` passed to it with the current system date. `setdate` is passed a similar structure containing the date which you wish to set. The structure `date` has the type:

```
struct date {
   short da_year;              year
   char da_day;                day
   char da_mon;                month
};
```

RETURNS

None.

SEE

`gettime, settime, stime, Tgetdate, Tsetdate`

SYNOPSIS

```
#include <ext.h>

getdfree(drive,info);

int drive;                  drive code, 0 => current
struct dfree*info;          disk information
```

DESCRIPTION-

This function obtains information about the specified disk drive, including the amount of free space available. If a 0 is passed as the drive number, information is obtained about the current drive. The `dfree` structure is defined in ext.h as follows:

```
struct dfree {
   unsigned long df_avail;     number of free clusters
   unsigned long df_total;     clusters per drive
   unsigned long df_bsec;      bytes per sector
   unsigned long df_sclus;     sectors per cluster
};
```

RETURNS

None.

getdisk, setdisk **Get or set current disk drive**

SYNOPSIS

```
#include <ext.h>

drive = getdisk();
bmap = setdisk(drive);

int drive;                  drive code
int bmap;                   bitmap of mounted drives
```

DESCRIPTION

The `setdisk` function changes the current drive code. Drive code 0 corresponds to drive A, code 1 is drive B and so on.

The getdisk function gets the current drive code, using the same codes as setdisk.

RETURNS

The function setdisk returns a bitmap of mounted drives, bit 0 corresponds to drive A, bit 1 is drive B and so on.

The function getdisk returns the code of the currently selected drive.

SEE

chgdsk, Dsetdrv, Dgetdrv, getcd, getdsk

getftime, setftime — Get/set file time/date

SYNOPSIS

```
#include <ext.h>

err = getftime(fh,curtime);    get file time
err = setftime(fh,newtime);    set file time

int err;                    0 if successful
int fh;                     handle of file
struct ftime *curtime;      pointer to current time
struct ftime *newtime;      pointer to time to be set
```

DESCRIPTION

These functions manipulate the time and date stamp of the file referenced by fh. getftime fills in the structure of type ftime passed to it with the current file time. settime is passed a similar structure containing the time which you wish to set. The structure ftime has the type:

```
struct ftime {
   unsigned ft_hour:5;        hour
   unsigned ft_min:6;         minute
   unsigned ft_tsec:5;        seconds
   unsigned ft_year:7;        year - 1980
   unsigned ft_month:4;       month
   unsigned ft_day:5;         day
};
```

RETURNS

If the operation is successful, a value of 0 is returned. Otherwise, the return value is -1, and further error information can be found in `errno` and `_OSERR`.

SEE

`chgft, Fdatime, getft`

gettime, settime — Get/set system time

SYNOPSIS

```
#include <ext.h>

gettime(curtime);          get current system time
settime(newtime);          set current system time

struct time *curtime;      pointer to current time
struct time *newtime;      pointer to time to be set
```

DESCRIPTION

These functions manipulate the system time. `gettime` fills in the structure of type `time` passed to it with the current system time. `settime` is passed a similar structure containing the time which you wish to set. The structure `time` has the type:

```
struct time {
   unsigned char ti_min;      minute
   unsigned char ti_hour;     hour
   unsigned char ti_hund;     hundredths - always 0
   unsigned char ti_sec;      seconds
};
```

RETURNS

None.

SEE

`getdate, setdate, stime, Tgetdate, Tsetdate`

ftw.h

ftw.h contains only one function, `ftw()`. This function enumerates the members of a directory calling a user defined function for each member.

ftw **Walk a file tree**

SYNOPSIS

```
#include <ftw.h>

err = ftw(root,fn,maxdir);

int err;                        error status
const char *root;               root of directory tree
int (*fn)(path,stat,type);      function called
int maxdir;                     maximum directory depth
const char *path;               path of current file
struct stat *stat;              pointer to stat block
int type;                       type of file encountered
```

DESCRIPTION

`ftw` is used to recursively traverse a directory structure. The search starts at `path` with the function `fn` called for every file/directory encountered. `maxdir` gives the maximum number of `opendir` calls which will be made, if the depth of the tree exceeds this value then the performance of the routine will be considerably reduced.

The string passed to `fn`, `path`, gives the path of the file currently being considered, whilst `stat` contains a pointer to a `struct stat` giving information about the file (see `stat`). The `type` parameter gives further information on the file:

Symbol	Meaning
FTW_F	normal file
FTW_D	directory
FTW_DNR	unreadable directory
FTW_NS	unreadable file info

Note that if the value is `FTW_NS` then the `stat` struct will not contain useful information.

The value returned by `fn` decides whether the tree walk is to continue or be terminated; the function should return zero to continue the traversal, or non-zero to terminate.

Note that `ftw` visits a directory before visiting any of its descendants.

RETURNS

The function returns the value from the last call to the user function `fn`, i.e. zero on success.

SEE

`opendir, stat`

EXAMPLE

```
#include <ftw.h>

int fn(const char *path, struct *x, int y)
{
   puts(path);
   return 0;        /* to continue the traversal */
}

int main(void)
{
   return ftw("",fn,20);
}
```

ieeefp.h

The ieeefp.h header file includes definitions for manipulating the math co-processors of the Mega ST^{E} and TT. These functions and variables allow the precision and rounding modes of the chips to be set and examined.

_FPCfpcr Floating point co-processor configuration

SYNOPSIS

```
#include <ieeefp.h>

long _FPCfpcr;              co-processor configuration
```

DESCRIPTION

The `_FPCfpcr` variable is used at program startup time to initialise the rounding and precision of the maths co-processor. The value is formed by ORing the required flags together. The flags are:

Symbol	Meaning
FP_RN	round to nearest
FP_RZ	round toward zero
FP_RM	round toward minus infinity
FP_RP	round toward minus infinity
FP_PX	extended precision
FP_PS	single precision
FP_PD	double precision
FP_X_BSUN	branch/set on unordered exception
FP_X_SNAN	signalling not a number exception
FP_X_OPERR	operand error exception
FP_X_OVFL	overflow exception
FP_X_UNFL	underflow exception
FP_X_DZ	divide by zero exception
FP_X_INEX2	inexact operation exception
FP_X_INEX1	inexact decimal input exception

Note that you should set only one of `FP_RN`, `FP_RZ`, `FP_RM` and `FP_RP`, equally you should set only one of `FP_PX`, `FP_PS` and `FP_PD`.

Note that changing this variable using anything other than a global initialisation will have no effect.

The default value of this variable is `FP_RN|FP_PX`. Note that if you enable any of the exceptions you must install a suitable exception handler.

SEE

```
fpgetround, fpsetround, fpgetprecision, fpsetprecision,
fpgetmask, fpsetmask
```

_FPCmode — Current math mode

SYNOPSIS

```
#include <ieeefp.h>

int _FPCmode;              maths evaluation package
```

DESCRIPTION

The `_FPCmode` variable is initialised by the startup code to indicate how floating point maths is performed. The following values are used:

Value	Meaning
1	Co-processor 68881 installed
2	Co-processor 68882 installed
4	68040 installed
-1	I/O based 68881 installed
0	No maths co-processor present

This variable is consulted by the auto-detecting maths routines to decide on how evaluation should proceed.

Your program may wish to interrogate this variable at program startup to ensure that sufficient maths hardware is available for your application, rather than have the user suffer a mysterious crash later on!

SYNOPSIS

```
#include <ieeefp.h>

typedef enum fp_except fp_except;

mask = fpgetmask();          get exception mask
mask = fpsetmask(ne);        set exception mask

fp_except mask;              old exception mask
fp_except ne;                new exception mask
```

DESCRIPTION

The `fpgetmask` and `fpsetmask` functions manipulate the exception mask of the maths co-processor. `fpgetmask` returns the current setting for this, whilst `fpsetmask` changes the exception mask to the value indicated by `ne`, returning the old value. The values used are:

Symbol	Meaning
FP_X_BSUN	branch/set on unordered
FP_X_SNAN	signalling not a number
FP_X_OPERR	operand error
FP_X_OVFL	overflow
FP_X_UNFL	underflow
FP_X_DZ	divide by zero
FP_X_INEX2	inexact operation
FP_X_INEX1	inexact decimal input

Note that these functions have no effect when the software IEEE emulation is being used, also beware that the standard runtime libraries include no support for dealing with exceptions, hence if you enable any you *must* install a suitable exception handler.

RETURNS

`fpgetmask` returns the current exception mask. `fpsetmask` returns the old exception mask.

SEE

_FPCmode, fpgetsticky, fpsetsticky

fpgetprecision, fpsetprecision — Get/set precision

SYNOPSIS

```
#include <ieeefp.h>

typedef enum fp_prec fp_prec;

prec = fpgetround();        get current precision
prec = fpsetround(np);      set new precision

fp_prec prec;               old precision
fp_prec np;                 new precision
```

DESCRIPTION

The fpgetprecision and fpsetprecision functions manipulate the precision setting of the maths co-processor. fpgetprecision returns the current setting for this, whilst fpsetprecision changes the precision to the value indicated by np, returning the old value. The values used are:

Symbol	Meaning
FP_PX	extended precision
FP_PS	single precision
FP_PD	double precision

Note that these functions have no effect when the software IEEE emulation is being used.

RETURNS

fpgetround returns the current precision. fpsetround returns the old precision setting.

SEE

fpgetround, fpsetround

SYNOPSIS

```
#include <ieeefp.h>

typedef enum fp_rnd fp_rnd;

rnd = fpgetround();          get current rounding mode
rnd = fpsetround(nr);        set new rounding mode

fp_rnd rnd;                  old rounding mode
fp_rnd nr;                   new rounding mode
```

DESCRIPTION

The `fpgetround` and `fpsetround` functions manipulate the rounding bits of the maths co-processor. `fpgetround` returns the current setting for this, whilst `fpsetround` changes the rounding to the mode indicated by `nr`, returning the old value. The values used for these modes are:

Symbol	Meaning
FP_RN	round to nearest
FP_RZ	round toward zero
FP_RM	round toward minus infinity
FP_RP	round toward minus infinity

Note that these functions have no effect when the software IEEE emulation is being used.

RETURNS

`fpgetround` returns the current rounding mode. `fpsetround` returns the old rounding mode.

SEE

`fpgetprecision, fpsetprecision`

SYNOPSIS

```
#include <ieeefp.h>

typedef enum fp_except fp_except;

prec = fpgetsticky();      get accrued exception byte
prec = fpsetsticky(ne);    set accrued exception byte

fp_except prec;            old accrued exception byte
fp_except ne;              new accrued exception byte
```

DESCRIPTION

The `fpgetsticky` and `fpsetsticky` functions manipulate the accrued exception byte of the maths co-processor. `fpgetsticky` returns the current set of accrued exceptions, whilst `fpsetsticky` changes the current value to `ne`. Note that these functions are 'sticky' because the co-processor never clears any of the bits, hence you may zero the word prior to a calculation and then afterwards collect the total of the problems which occurred. The values used to represent the exception conditions are:

Symbol	Meaning
FP_X_BSUN	branch/set on unordered
FP_X_SNAN	signalling not a number
FP_X_IOP	invalid operation
FP_X_OVFL	overflow
FP_X_UNFL	underflow
FP_X_DZ	divide by zero
FP_X_INEX	inexact operation

Note that these functions have no effect when the software IEEE emulation is being used.

RETURNS

`fpgetsticky` returns the current accrued exception byte. `fpsetsticky` returns the old accrued exception byte.

SEE

fpgetmask, fpsetmask

osbind.h

With the advent of the TT and Mega-ST^{E} two new TOSes are available, 2.xx and 3.xx. To cater for these new machines additional OS calls are available in osbind.h.

Bconmap — Get/Set AUX: device mapping

SYNOPSIS

```
#include <osbind.h>

val = Bconmap(devno);

long val;                    return value
int devno;                   BIOS device number
```

DESCRIPTION

Bconmap is used to control the mapping of the AUX: device (BIOS device 1) which is initially set to the ST compatible serial port. Valid device assignments on the TT are:

devno	Meaning
-2	Return pointer to struct bconmap
-1	Read existing setting
6	Select ST-compatible serial port
7	Select modem 2 (SCC channel B)
8	Select serial 1 (3-wire TT MFP)
9	Select serial 2 (SCC channel A)

Note that these device assignments are *specific* to the TT; other hardware will use different assignments.

RETURNS

As noted above.

SEE

Bconin, Bconout, Bconis, Bconos, Iorec, Rsconf

DMAread — Read sectors from DMA device

SYNOPSIS

```
#include <osbind.h>

err = DMAread(sector, count, buffer, devno);

long err;                   error status
long sector;                first sector to read
int count;                  number of sectors to read
void *buffer;               pointer to buffer
int devno;                  DMA device id
```

DESCRIPTION

DMAread is used to read count sectors from the DMA device given by devno starting at sector into memory. The values of devno are:

devno	Meaning
0 - 7	ACSI devices 0-7
8 - 15	SCSI devices 0-7

buffer is a pointer to a suitable memory block. Note that reads from ACSI devices *must* specify a memory block in system RAM. If a transfer to alternative RAM is required it may be possible to use the memory pointed to by _FRB *provided* the _flock system variable is suitably managed.

RETURNS

DMAread returns 0 normally, or a non-zero error code.

SYNOPSIS

```
#include <osbind.h>

err = DMAwrite(sector, count, buffer, devno);

long err;                   error status
long sector;                first sector to write
int count;                  number of sectors to write
const void *buffer;         pointer to buffer
int devno;                  DMA device id
```

DESCRIPTION

DMAwrite is used to write count sectors from the DMA device given by devno starting at sector from memory. The values of devno are:

devno	Meaning
0 - 7	ACSI devices 0-7
8 - 15	SCSI devices 0-7

buffer is a pointer to a suitable memory block. Note that writes to ACSI devices *must* specify a memory block in system RAM. If a transfer from alternative RAM is required it may be possible to use the memory pointed to by _FRB *provided* the _flock system variable is suitably managed.

RETURNS

DMAread returns 0 normally, or a non-zero error code.

SYNOPSIS

```
#include <osbind.h>

EgetPalette(num, count, palette)

int num;                    initial CLUT entry to modify
int count;                  number of CLUT entries
short *palette;             pointer to new CLUT entries
```

DESCRIPTION

`EgetPalette` copies the contents of a contiguous set of the TT CLUT registers into the array pointed to by `palette`. `num` gives the first entry to copy, whilst `count` gives the number of entries. The CLUT entries are encoded in the following manner:

bits 15-12	bits 11-8 (Red)				bits 7-4 (Green)				bits 3-0 (Blue)			
Unused	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

`R0` represents the least-significant bit of the red component of the colour, `R3` the most-significant. Similarly, `G0-G3` give the green component and `B0-B3` the blue component. Note that this (and the other TT specific palette calls) do *not* use the ST compatible method of encoding the colour as per `Setpalette`.

RETURNS

None.

SEE

`Setpalette, Setcolor, EsetColor, EsetPalette`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the _VDO cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

mode = EgetShift()

int mode;                    video shift register value
```

DESCRIPTION

The `EgetShift` call returns the current setting of the TT video shifter. The meaning of the bits within the value are:

Bit 15	Bit 12	Bits 10-8		Bits 3-0
Smear mode	Grey mode	000	STlow	Current colour bank
		001	ST medium	
		010	ST high	
		100	TT medium	
		110	TT high	
		111	TT low	

RETURNS

The call returns the current value of the video shifter.

SEE

`Setscreen, Getrez, EsetSmear, EsetShift, EsetGray, EsetBank`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the _VDO cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

old = EsetBank(new)

int old;                    old active colour bank
int new;                    colour bank to select
```

DESCRIPTION

`EsetBank` selects which of the 16 CLUTs (colour lookup table) is active. If the value passed is negative the active CLUT is not changed and the old value is simply returned.

RETURNS

The call returns the old active CLUT number.

SEE

`Setpalette, EsetPalette, EgetPalette`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the _VDO cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

old = EsetColor(num, color)

int old;                    old TT CLUT value
int num;                    TT CLUT entry to modify
int color;                  new colour value
```

DESCRIPTION

`EsetColor` sets the absolute colour entry `num` to the value given by `color`. The encoding used for `color` is as follows:

bits 15-12	bits 11-8 (Red)				bits 7-4 (Green)				bits 3-0 (Blue)			
Unused	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

`R0` represents the least-significant bit of the red component of the colour, `R3` the most-significant. Similarly, `G0-G3` give the green component and `B0-B3` the blue component. Note that this (and the other TT specific palette calls) do *not* use the ST compatible method of encoding the colour as per `Setcolor`.

If `color` is negative then the colour is not changed and the old value is returned.

RETURNS

The old value of the TT CLUT entry is returned.

SEE

`Setcolor, EsetPalette, EgetPalette`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the `_VDO` cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

old = EsetGray(new);

int old;                    old grey scale mode
int new;                    new grey scale mode
```

DESCRIPTION

`EsetGray` is used to read/write the TT video hardware's grey mode bit. When grey mode is set, the bottom eight bits of the palette value are used as one of 256 possible grey levels. `new` is zero to select colour mode, +ve to select grey mode, or -ve to simply return the old setting.

RETURNS

The old grey scale value.

SEE

`EsetShift, EgetShift`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the _VDO cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

EsetPalette(num, count, palette)

int num;                    intial CLUT entry to modify
int count;                  number of CLUT entries
short *palette;             pointer to new CLUT entries
```

DESCRIPTION

`EsetPalette` sets the contents of a contiguous set of the TT CLUT registers. `num` gives the first entry to modify, whilst `count` gives the number of entries. `palette` is a pointer to an array of `count` CLUT entries encoded in the following manner:

bits 15-12	bits 11-8 (Red)				bits 7-4 (Green)				bits 3-0 (Blue)			
Unused	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

`R0` represents the least-significant bit of the red component of the colour, `R3` the most-significant. Similarly, `G0-G3` give the green component and `B0-B3` the blue component. Note that this (and the other TT specific palette calls) do *not* use the ST compatible method of encoding the colour as per `Setpalette`.

RETURNS

None.

SEE

`Setpalette, EsetColor, EgetPalette`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the `_VDO` cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

old = EsetShift(new)

int old;                    old shift mode register value
int new;                    old shift mode register value
```

DESCRIPTION

The `EsetShift` call is used to change the setting of the TT video shifter. The meaning of the bits within the values are:

Bit 15	Bit 12	Bits 10-8		Bits 3-0
Smear mode	Grey mode	000	ST low	Current colour bank
		001	ST medium	
		010	ST high	
		100	TT medium	
		110	TT high	
		111	TT low	

RETURNS

The call returns the old value of the video shifter.

SEE

`Setscreen, Getrez, EsetSmear, EgetShift`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the _VDO cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

old = EsetSmear(new);

int old;                    old smear mode
int new;                    new smear mode
```

DESCRIPTION

`EsetSmear` is used to read/write the TT video hardware's smear mode bit. When smear mode is set, the video hardware displays video pixels with value 0 as the last non-zero colour rather than colour zero itself. This can be used to change the colour of a filled-polygon by only changing its outline rather than via a complete re-fill. `new` is zero to select colour mode, +ve to select grey mode, or -ve to simply return the old setting.

RETURNS

The old smear value.

SEE

`EsetShift, EgetShift`

CAVEATS

This function requires that the program is running on a TT. Applications which use this call should check the high word of the `_VDO` cookie for the value 2 before attempting it.

SYNOPSIS

```
#include <osbind.h>

res = Getrez();

int res;                    current screen mode
```

DESCRIPTION

`Getrez` returns a coded value for the current screen mode. The values *currently* returned in `res` are:

Value	Screen mode
0	Low resolution (320x200x4)
1	Medium resolution (640x200x2)
2	High resolution (640x400x1)
4	TT medium resolution (640x480x4)
6	TT high resolution (1280x1024x1)
7	TT low resolution (320x480x8)

RETURNS

As noted above.

SEE

`v_opnwk, Setscreen`

CAVEATS

You should *not* use this function except as indicated under `v_opnvwk`. If you do rely on this function your application will, in general, not work on large screen monitors or on future extended screen modes.

If your application needs to know the size of the screen, the number of bitplanes, or other mode specific information it should interrogate the AES or VDI for the information rather than relying on hard-coded constants based on the result of this call.

Maddalt — *Inform GEMDOS of alternative memory*

SYNOPSIS

```
#include <osbind.h>

err = Maddalt(start, size);

int err;                    0 or error code
void *start;                start of memory block
long size;                  size of memory block
```

DESCRIPTION

This call is used to inform GEMDOS of the presence of alternative RAM; it should not be needed unless you have added custom memory to the system which is not detected by the BIOS at boot time. `start` is pointer to the block of memory, whilst `size` gives the number of blocks in the block.

RETURNS

`Maddalt` returns 0 for success, or a GEMDOS error code.

SEE

`Mxalloc`

Mxalloc — *Allocate block of from preferred pool*

SYNOPSIS

```
#include <osbind.h>

base = Mxalloc(amount, mode);

void *base;                 base of block allocated
long amount;                amount of memory requested
int mode;                   memory type preference
```

DESCRIPTION

The `Mxalloc` function is used to obtain blocks of memory from a preferred GEMDOS free memory pool.

The amount of memory required is passed in `amount`, and the base of the block allocated is returned in `base`. If no memory is available a `NULL` pointer is returned. The `mode` parameter gives the type of memory the application is interested in receiving and has the values:

Mode	Meaning
0	ST RAM only
1	alternative RAM only
2	either, ST RAM preferred
3	either, alternative preferred.

To determine the size of the largest free block in the system of a given type, the value -1 may be used for `amount`, when the pointer returned should be cast to a long value giving the size of the block. Note that it is the size of the largest free block that is returned, and *not* the total free memory in the OS pool.

RETURNS

`Mxalloc` returns the base of the memory block to use or `NULL` if insufficient memory was available. If `amount` is equal to -1 then the size of the largest block is returned.

SEE

`Mfree, Mshrink, Malloc`

CAVEATS

This call was added in TOS 2.0. An application may detect the presence of alternative memory (aka fast memory) by interrogating the `_FRB` cookie, which will only be present on machines with alternative RAM.

SYNOPSIS

```
#include <osbind.h>

err = NVMaccess(op, start, count, buffer);

int err;                    error status
int op;                     operation to perform
int start;                  first location to read/write
int count;                  number of bytes to read/write
char *buffer;               pointer to buffer
```

DESCRIPTION

`NVMaccess` is used to access the non-volatile memory in the Atari TT's real time clock. `op` gives the operation which is to be performed and has the following values:

op	Meaning
0	Read NVM data
1	Write NVM data
2	Reset and initialise NVM

`start` gives the first of `count` bytes which should be read/written from/to `buffer`.

RETURNS

`NVMaccess` returns 0 normally, or a non-zero error code.

CAVEATS

At the time of writing no public entries within the non-volatile memory have been committed by Atari. *All* locations are reserved for use by Atari.

SYNOPSIS

```
#include <osbind.h>

error = Pexec(mode,path,tail,env);

long error;                 error return
short mode;                 Pexec mode
const char *path;           path of program to execute
const char *tail;           command line
const char *env;            pointer to environment
```

DESCRIPTION

`Pexec` provides facilities for a program to create basepages, load programs and execute them.

`path` is a pointer a string giving the filename of the program to execute. If `path` does not specify a drive the current drive is used, similarly if no pathname is specified the current path is used. Note that any filename extension must be explicitly specified.

`tail` is a pointer to a length prefixed string, i.e. `tail[0]` contains the length of the string starting at `tail[1]`, the total length of the string (including the length byte) may not exceed 126 bytes. Note that when copying this string GEMDOS copies 126 bytes or up to a NUL character, which ever is first.

`env` contains a pointer to the environment to be passed to the child process. If this pointer is `NULL` then the child inherits a copy of the parents environment. GEMDOS obtains a block of memory using `Malloc` into which it copies the child processes environment.

The `mode` parameter determines what function the command performs. The following `mode` values are allowed:

Value	Meaning
0	Create a basepage, load program into the basepage, execute program returning program's termination code when the program completes.
3	Create a basepage and load program into it. The value returned is the address of the base page created.

4	Execute program already loaded. For this mode `path` and `env` are unused (pass `NULL` for these). `tail` holds the address of the program to execute. The value returned is the program termination code. Note that the TPA and environment are not freed after running the program.
5	Create a basepage. For this mode `path` is unused (pass `NULL` for this), `tail` and `env` have there normal meanings. The value returned is the address of the base page created.
6	Execute program already loaded. For this mode `path` and `env` are unused, and `tail` holds the address of the program to execute. The value returned is the program termination code. Unlike mode 4, the TPA and environment are freed after executing the child process. Note the warning below about this mode.
7	Create a basepage. For this mode `path` holds the program load flags as set in the executable file's load bits. `tail` and `env` have their normal meanings. The value returned is the address of the base page created. Note the warning below about this mode.

Note that the `basepage` structure is described in the C library manual and also in the `basepage.h` header file.

RETURNS

`Pexec` returns values dependent on the `mode` argument. For all modes a *longword* negative value is an error indication, positive values are as indicated above. Note that when `Pexec` returns an exit code from a program it has executed the top 16 bits are zero, you may also find it useful to note that if a program is aborted via `Ctrl-C` then the return code is 0xffe0.

SEE

`Pterm0, Pterm, Ptermres, Mshrink`

CAVEATS

`Pexec` mode 6 is only available on GEMDOS version 0.21 (TOS 1.4) and above. `Pexec` mode 7 is only available on GEMDOS version 0.24 (TOS 2.0) and above.

getopt — Get option letter from argument vector

SYNOPSIS

```
#include <stdlib.h>

c = getopt(argc,argv,optstring);

int c;                      argument character
int argc;                   argument count
const char *argv[];         argument vector
const char *optstring;      string containing valid opts

extern char *optarg;        pointer to option argument
extern int optind;          index of next argument
extern int opterr;          error message setting
```

DESCRIPTION

The `getopt` function returns the next option letter in `argv` which matches a letter in `optstring`. `optstring` contains all the option letters which are to be recognised, optionally followed by a colon (`:`) when an argument is required by the option. Such an argument may either be concatenated with the option letter, or be the next argument. The external variable `optarg` is set to point to any such argument. If the colon is doubled (`::`) then the argument is optional and if present *must* be concatenated to the option letter, if no argument was present then `optarg` will be `NULL`.

The external variable `optind` is used to track the next `argv` index which `getopt` will use and is normally initialised to 1 by the first call to `getopt`.

When all options have been processed (i.e. the first argument which does not start with a '-'), or the special delimiter '--' has been encountered the value -1 is returned and and the '--' argument skipped.

When an unrecognised option is encountered, or an argument option is omitted where one was expected, an error message is printed on `stderr` and the value '?' returned. The printing of error messages may be disabled by setting the external variable `opterr` to 0.

Note that unlike `argopt`, `getopt` does *not* recognise a '/' as an option prefix.

RETURNS

The value of the character obtained as an option, '?' for an invalid option or -1 if no more arguments are available.

SEE

`argopt, main`

EXAMPLE

```
/*
 * parse the command lines:
 *    myprog -x -ypdq -z -g moo blah
 */

#include <stdlib.h>

int main(int argc, char *argv[])
{
   int c;
   char *file,*status = NULL;
   int x=0,z=0;

   while ((c=getopt(argc,argv,"xy::zg:"))!=-1)
      switch (c) {
         case 'x':
            x++;
            break;

         case 'z':
            z++;
            break;

         case 'y':
            if (optarg)
               status=optarg;
            break;

         case 'g':
            file=optarg;
            break;
```

```
        case '?':
          abort();
          break;
      }

   for (; optind<argc; optind++)
      process(argv[optind],x,z,status,file);

   return 0;
}
```

spawn ***Launch new process***

SYNOPSIS

```
#include <stdlib.h>

error = spawnl(mode,prog,arg0,…,argn,NULL);
error = spawnv(mode,prog,argv);

error = spawnle(mode,prog,arg0,…,argn,NULL,envp);
error = spawnve(mode,prog,argv,envp);

error = spawnlp(mode,prog,arg0,…,argn,NULL);
error = spawnvp(mode,prog,argv);

error = spawnlpe(mode,prog,arg0,…,argn,NULL,envp);
error = spawnvpe(mode,prog,argv,envp);

int error;                  error code
int mode;                   execution mode of program
const char *prog;           program name
const char *arg0;           argument #0
const char *argn;           argument #n
const char *argv[];         argument vector
const char *envp[];         environment pointers

extern int _aecl;           Extended command lines flag
```

DESCRIPTION

The spawn... family of functions provide the most control over the creation of processes. The mode parameter specifies how the named program is to be launched.

The values of this are:

Symbol	Meaning
P_WAIT	Spawn process and wait for child to terminate, returning termination code.
P_NOWAIT	Spawn process concurrently, termination code available from wait. This is equivalent to fork...
P_OVERLAY	Overlay current process. Does not return. This is equivalent to exec...

Note that P_NOWAIT is not supported by GEMDOS, so the calling process actually waits for the child to terminate before continuing. Also the P_OVERLAY mode is not available from GEMDOS and is simulated by the spawn... family function terminating via _exit after execution; this mode is designed for use with vfork.

Details on the various modes of this command are as under the fork... family of functions.

RETURNS

The return code is as specified above for a successful execution, otherwise the specified program file cannot be found, a -1 return is made, and additional error information can be found in errno and _OSERR.

SEE

Pexec, _exit, exec..., fork..., vfork, wait

string.h is the ANSI string definitions file. To support other pre-ANSI platforms some additional string and memory functions are available.

bcmp, bcopy, bzero — BSD memory block operations

SYNOPSIS

```
#include <string.h>

x = bcmp(a,b,n);          Compare two memory blocks
s = bcopy(from,to,n);     Copy a memory block
s = bzero(to,n);          Zero a memory block

void *to;                 destination pointer
const void *from;         source pointer
size_t n;                 number of bytes
const void *a,*b;         block pointers
void *s;                  return pointer
int x;                    return value
```

DESCRIPTION

These functions manipulate blocks of memory in various ways. The `bcmp` function is identical to the `memcmp` function and compares two blocks of memory. The `bzero` function zeroes the nominated block of memory; note that unlike the `memset` function it is not possible to change the value which is used for the initialisation. The `bcopy` function copies the area from one location to the other in the same manner as the ANSI `memcpy` function; note that `from` and `to` are transposed with respect to `memcpy`.

RETURNS

As noted above.

SEE

`memcmp, memcpy, memset`

SYNOPSIS

```
#include <string.h>

p = index(s,c);          find character in string
p = rindex(s,c);         find last character in string

char *p;                 updated string pointer
const char *s;           input string pointer
int c;                   character to be located
```

DESCRIPTION

The `index` function scans the input string to find the first occurrence of the character specified by argument `c`. The `rindex` function scans the input string to find the last occurrence of the character specified by argument `c`.

Note that these functions are almost identical to the ANSI `strchr` and `strrchr` functions, however these functions *never* match the trailing `\0`.

RETURNS

A `NULL` pointer is returned if the input string is empty or if the specified character is not found.

fstat — Get status of file handle

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

ret = fstat(fd,statbuf);

int ret;                    0 if successful
int fd;                     file handle of file
struct stat *statbuf;       stores information about file
```

DESCRIPTION

The `fstat` function returns UNIX-style file status information about the file specified by `fd`. The buffer returned is defined in sys/stat.h as follows:

```
struct stat {
    dev_t st_dev;               disk drive number
    ino_t st_ino;               inode number (not used)
    unsigned short st_mode;     file mode flags
    short st_nlink;             number of links (always 1)
    short st_uid;               user id (not used)
    short st_gid;               group id (not used)
    dev_t st_rdev;              same as st_dev
    off_t st_size;              file size in bytes
    time_t st_atime;            time of last access
    time_t st_mtime;            time of last modification
    time_t st_ctime;            time of creation
};
```

Note that the header file sys/types.h must be included prior to sys/stat.h as this defines the types `dev_t`, `ino_t`, `dev_t` and `off_t`.

RETURNS

On success, the `fstat` function returns 0.

SYNOPSIS

```
#include <fcntl.h>

old = umask(mask)

int old;                    existing mask setting
int mask;                   desired setting for file mask
```

DESCRIPTION

`umask` sets the file mode creation mask to its argument and returns the previous value of the mask. The default value is 0 (all permissions available). In practice this function is only useful when called as `umask(S_IWRITE)` to create read only files by default, since that is the only UNIX® style protection bit implemented by GEMDOS.

RETURNS

The old value of the file mask is returned

SEE

`chmod, creat, Fcreat, open`

stime — Set current system time

SYNOPSIS

```
#include <time.h>

err = stime(time)

int err;                    error return
const time_t *time;         pointer to time
```

DESCRIPTION

The `stime` function changes the current time and date in the system clock to the value pointed to by `time`.

RETURNS

The function returns 0 on successfully changing the time, or -1 to indicate an error, with further information in `errno`.

SEE

`Tsetdate, Tsettime, time, utime`

exec — Overlay current process

SYNOPSIS

```
#include <unistd.h>

error = execl(prog,arg0,arg1,…,argn,NULL);
error = execv(prog,argv);

error = execle(prog,arg0,arg1,…,argn,NULL,envp);
error = execve(prog,argv,envp);

error = execlp(prog,arg0,arg1,…,argn,NULL);

error = execvp(prog,argv);

error = execlpe(prog,arg0,arg1,…,argn,NULL,envp);
error = execvpe(prog,argv,envp);

int error;                  error code
const char *prog;           program name
const char *arg0;           argument #0
const char *arg1;           argument #1
const char *argn;           argument #n
const char *argv[];         argument vector
const char *envp[];         environment pointers

extern int _aecl;           extended command lines flag
```

DESCRIPTION

These functions mimic the UNIX® process overlay commands. GEMDOS does not actually support this, so the implementation spawns a child, terminating (via `_exit`) on return. This family of functions are designed for use with the `vfork` function.

Details on the various modes of this command are as under the `fork...` family of functions.

RETURNS

If the call succeeds the function never returns. If the specified program file cannot be found, a -1 return is made, and additional error information can be found in `errno` and `_OSERR`. Note that you must call the `wait` function in order to obtain the completion code from the child process.

SEE

`Pexec, _exit, fork, spawn, vfork, wait`

vfork — Spawn new process

SYNOPSIS

```
#include <unistd.h>

pid = vfork();

int pid;                    process id of child or zero
```

DESCRIPTION

`vfork` is used to create new processes. Under UNIX `fork` creates a new concurrent process, `vfork` simulates this by 'borrowing' the parents address space until a call to `exec...` is made.

`vfork` returns 0 in the child's context and (later) the process id of the child in the parent's context.

When using `vfork` you must be careful that you do not alter the context of the parent process in the child. The child process must not return from the function which called `vfork`, if this were to happen the parent process would then return to a stack frame which had been deallocated by the child.

If the call to `exec...` should fail then the child *must* call `_exit`, rather than `exit` which would otherwise close the standard I/O files.

RETURNS

`vfork` returns 0 in the child's context and (later) the process id of the child in the parent's context.

SEE

exec…, fork…, spawn…, wait

EXAMPLE

```
#include <unistd.h>
#include <stddef.h>

int main(int argc,char *argv[])
{
   if (!vfork())
      execl(argv[1],argv[1],NULL);
   else
      execl(argv[2],argv[2]),NULL);
}
```

vdi.h

vdi.h includes all the binding routines which are available for call the GEM VDI. There are many new functions available for use either with GDOS or FSM-GDOS.

v_pgcount — Set number of copies for laser printer

SYNOPSIS

```
#include <vdi.h>

v_pgcount(handle,n);

int handle;              device handle
int n;                   number of additional copies
```

DESCRIPTION

v_pgcount is used to change the number of copies generated by laser printer driver from the default of 1. The parameter, n, gives the number of *additional* copies which are required.

This function requires that GDOS, Font-GDOS or FSM-GDOS is loaded.

RETURNS

None.

SEE

`v_clear_disp_list, v_clrwk, v_opnwk`

vq_extnd — Extended Inquire

SYNOPSIS

```
#include <vdi.h>

vq_extnd(handle,flag,work_out);

int handle;              workstation handle
int flag;                0=normal; 1= extended inquire
short *work_out;         values returned
```

DESCRIPTION

This function can be used to return the information returned by the `v_opnwk` or `v_opnvwk` calls (if `flag==0`) or additional values if `flag==1`. The `work_out` array must have room for at least 57 shorts. The values returned when `flag=0` are detailed under `v_opnwk`. The values returned when `flag==1` are as follows:

`work_out[0]`	Type of screen: 0 = not screen. 4 = 'normal' screen with common graphics and character memory. Other values are not applicable to the ST.
`work_out[1]`	Number of background colours available.
`work_out[2]`	Text effects supported. See `vst_effects`.
`work_out[3]`	Scaling of rasters: 0 = scaling not supported. 1 = scaling supported.
`work_out[4]`	Number of planes available.
`work_out[5]`	Lookup table supported 0 = table supported. 1 = table not supported.

work_out[6]	Performance factor. Number of 16x16 pixel raster operations per second.
work_out[7]	Contour fill capability: 0 = no. 1 = yes.
work_out[8]	Character rotation ability: 0 = none. 1 = multiples of 90 degrees only. 2 = any angle.
work_out[9]	Number of writing mode available.
work_out[10]	Highest level of input mode available: 0 = none. 1 = request. 2 = sample.
work_out[11]	Text alignment capability flag: 0 = no. 1 = yes.
work_out[12]	Inking capability flag: 0 = no. 1 = yes.
work_out[13]	Rubber-banding capability flag: 0 = no. 1 = rubber-band lines possible. 2 = rubber-band lines and rectangles possible.
work_out[14]	Maximum vertices for polyline, polymarker or filled area (-1 = no maximum).
work_out[15]	Maximum index for intin (-1 = no maximum).
work_out[16]	Number of keys on the mouse.
work_out[17]	Styles available for wide lines: 0 = no. 1 = yes.
work_out[18]	Writing modes available for wide lines: 0 = no. 1 = yes.
work_out[19]	Clipping enabled flag: 0 = no. 1 = yes.
work_out[20] ... work_out[44]	Reserved.

work_out[45]	First X co-ordinate of clipping rectangle
work_out[46]	First Y co-ordinate of clipping rectangle
work_out[47]	Second X co-ordinate of clipping rectangle
work_out[48]	Second Y co-ordinate of clipping rectangle
work_out[49]	Reserved.
...	
work_out[56]	

SEE

v_opnwk, v_opnvwk

v_bez — Draw bezier curve

SYNOPSIS

```
#include <vdi.h>

v_bez(handle,count,xy,bezarr,extent,totpts,totmoves);

int handle;                 workstation handle
int count;                  number of vertices
const short *xy;            array of vertices
const char *bezarr;         array of vertex descriptors
short extent[4];            extent of resultant bezier
short *totpts;              total points in polygon
short *totmoves;            total moves in polygon
```

DESCRIPTION

v_bez is used to draw an unfilled bezier curve on the device referenced by handle. xy points to a set of (x, y) co-ordinate pairs, whilst bezarr lists the attributes of each of the corresponding points. The following bits are used in bezarr:

Bit	Value	Meaning
0	0	Point begins a polyline section.
0	1	Point is the first point of a set of 4 bezier points in the sequence: first anchor point, first control point, second control point, second anchor point.
1	1	Point is a jump point; the current point is moved to this point without a joining line

The total number of points in the resulting polygon is returned in `totpts`, whilst the total number of moves in the resulting polygon is returned in `totmoves`. The bounding box of the polygon is returned in `extent`.

This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`v_pline, v_bez_fill`

EXAMPLE

```
#include <aes.h>
#include <vdi.h>
#include <osbind.h>

short work_out[57];
short work_in[11]={0,1,1,1,1,1,1,1,1,1,2};

main()
{
   short h, junk;
   short pts[8],extent[4],totpts,totmoves,act;

   appl_init();
   work_in[0] = Getrez()+ 2;
   h = graf_handle(&junk, &junk, &junk, &junk);
   v_opnvwk(work_in, &h, work_out);
   v_bez_on(h);
   v_bez_qual(h, 100, &act);

   pts[0]=100;
   pts[1]=100;

   pts[2]=100;
   pts[3]=400;

   pts[4]=400;
   pts[5]=100;

   pts[6]=400;
   pts[7]=400;

   v_bez(h, 4, pts, "\1\0\0\0", extent, &totpts,
    &totmoves);
```

```
    v_bez_off(h);
    v_clsvwk(h);
    appl_exit();
}
```

v_bez_con — *Control GDOS bezier facilities*

SYNOPSIS

```
#include <vdi.h>

qual = v_bez_con(handle, onoff);

int qual;                   maximum bezier depth
int handle;                 workstation handle
int onoff;                  1 to enable, 0 to disable
```

DESCRIPTION

`v_bez_con` is used to enable or disable GDOS bezier capabilities. The `onoff` parameter is 0 to disable bezier facilities or 1 to enable them. Note that two macros are provided for this purpose, `v_bez_on()` and `v_bez_off()`.

Note that failure to disable bezier facilities prior to closing the (virtual) workstation may cause the VDI to crash.

This function requires that Font-GDOS or FSM-GDOS is loaded.

RETURNS

When enabling bezier operation a value, ranging from 0 to 7, is returned giving the maximum bezier depth, with 2^{qual} giving the number of line segments used to make up the curve.

SEE

`v_set_app_buff, v_bez, v_bezfill`

SYNOPSIS

```
#include <vdi.h>

v_bez_fill(handle,count,xy,bezarr,extent,totpts,
                          totmoves);

int handle;               workstation handle
int count;                number of vertices
const short *xy;          array of vertices
const char *bezarr;       array of vertex descriptors
short extent[4];          extent of resultant bezier
short *totpts;            total points in polygon
short *totmoves;          total moves in polygon
```

DESCRIPTION

`v_bez_fill` is used to draw a filled bezier curve on the device referenced by `handle`. `xy` points to a set of (`x`, `y`) co-ordinate pairs, whilst `bezarr` lists the attributes of each of the corresponding points. The following bits are used in `bezarr`:

Bit	Value	Meaning
0	0	Point begins a polyline section.
0	1	Point is the first point of a set of 4 bezier points in the sequence: first anchor point, first control point, second control point, second anchor point.
1	1	Point is a jump point; the current point is moved to this point without a joining line

The total number of points in the resulting polygon is returned in `totpts`, whilst the total number of moves in the resulting polygon is returned in `totmoves`. The bounding box of the polygon is returned in `extent`.

This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`v_pline, v_bez`

v_bez_qual — Set bezier quality

SYNOPSIS

```
#include <vdi.h>

v_bez_qual(handle, percent, actual);

int handle;              workstation handle
int percent;             speed/quality percentage
short *actual;           actual percentage selected
```

DESCRIPTION

`v_bez_qual` sets the bezier speed/quality trade-off parameter as a percentage of the quality (hence 100% is best quality but slowest). The quality factor is passed in `percent` as a value from 0 to 100. The value selected by GDOS (approximated to the 8 levels available) is returned in actual.

This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`v_bez, v_bez_fill, v_set_app_buff`

v_flushcache — Flush FSM font cache

SYNOPSIS

```
#include <vdi.h>

v_flushcache(handle);

int handle;              workstation handle
```

DESCRIPTION

The `v_flushcache` call empties the FSM font caches. This function requires that FSM GDOS is loaded.

SEE

`v_loadcache, v_savecache`

SYNOPSIS

```
#include <vdi.h>

v_ftext(handle,x,y,str);

int handle;              workstation handle
int x;                   x co-ordinate of start
int y;                   y co-ordinate of start
const char *str;         characters to output
```

DESCRIPTION

This function is identical to the normal VDI `v_gtext` function, however the text drawn takes into account the remainder values from `vqt_advance`, resulting in more accurate spacing.

v_getoutline *Get FSM outline*

SYNOPSIS

```
#include <vdi.h>

v_getoutline(handle, ch, component);

int handle;              workstation handle
int ch;                  character to obtain outline
fsm_component_t *cpt;    FSM component pointer
```

DESCRIPTION

This function requires FSM-GDOS for operation and obtains a pointer to the outline for the character `ch`.

A pointer to the character outline is returned in `component`. A description of the `fsm_component_t` data structure is beyond the scope of this document.

This function requires that FSM-GDOS is loaded.

SEE

`v_killoutline`

SYNOPSIS

```
#include <vdi.h>

v_killoutline(handle, component);

int handle;                 workstation handle
fsm_component_t *cpt;       FSM component pointer
```

DESCRIPTION

This function requires FSM-GDOS for operation and releases the memory occupied by the FSM component (obtained via `v_getoutline`).

This function requires that FSM-GDOS is loaded.

SEE

`v_getoutline`

v_loadcache **Load FSM font cache from disk**

SYNOPSIS

```
#include <vdi.h>

err = v_loadcache(handle,name,mode);

int err;                    non-zero if an error occurred
int handle;                 workstation handle
char *name;                 file name
int mode;                   0 to append, 1 to replace
```

DESCRIPTION

`v_loadcache` is used to restore a font cache previously saved using `v_savecache`. `name` specifies the file which is to be loaded, normally this should have a .FSM extension. `mode` specifies whether the file replaces or appends to the existing font cache. If the value is 0 the file is appended, otherwise the font cache is replaced.

This function requires that FSM-GDOS is loaded.

SEE

v_savecache, v_flushcache

v_savecache — Save FSM font cache to disk

SYNOPSIS

```
#include <vdi.h>

err = v_savecache(handle,name);

int err;                    non-zero if an error occurred
int handle;                 workstation handle
char *name;                 file name to save cache in
```

DESCRIPTION

v_savecache is used to save the font cache for reloading by v_loadcache. name specifies the file which in which the cache is to be saved, normally this should have a .FSM extension.

This function requires that FSM-GDOS is loaded.

SEE

v_loadcache, v_flushcache

v_set_app_buff — Reserve bezier workspace

SYNOPSIS

```
#include <vdi.h>

v_set_app_buff(buffer, npara);

void *buffer;               pointer to buffer
int npara;                  number of paragraphs
```

DESCRIPTION

This call makes the nominated memory block available for use by the GDOS bezier extensions. If this call is not made, a default 8K buffer is allocated by GDOS.

`buffer` is a pointer to the memory block, and `npara` is the number of *paragraphs* available in the block (a paragraph is 16 bytes of memory).

Note that *no workstation handle* is passed. This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`v_bez_con`

vq_vgdos — Obtain GDOS version number

SYNOPSIS

```
#include <vdi.h>

ver = vq_vgdos();

unsigned long ver;          GDOS revision marker
```

DESCRIPTION

This function indicates what version of (or whether) GDOS is loaded. GDOS is the part of GEM that was left out of the ST's ROMs; it provides the ability to load fonts from disk, load printer drivers and use device-independent co-ordinates.

The values returned by the function are:

Symbol	Meaning
`GDOS_FNT`	FONT GDOS is loaded. This is the same as FSM GDOS with the font scaling module removed.
`GDOS_FSM`	FSM GDOS is loaded. This indicates that all FSM functions are available.
`GDOS_NONE`	No GDOS is present. Note that there is no code to indicate a pre-FSM release of GDOS, this is indicated by any other value.

You should always use this function to determine whether GDOS is loaded, otherwise the system will crash if you use a facility not provided by the ROM (such as opening a physical workstation).

SEE

`v_opnwk, vq_gdos`

SYNOPSIS

```
#include <vdi.h>

vqt_advance(handle, ch, advx, advy, xrem, yrem);

int handle;              workstation handle
int ch;                  character
short *advx;             X advance value
short *advy;             X advance value
short *xrem;             X remainder (modulo 16384)
short *yrem;             Y remainder (modulo 16384)
```

DESCRIPTION

This function returns the X and Y advance values for the character `ch` in `advx` and `advy` respectively, together with any fractional advances in `xrem` and `yrem`. Note that this function takes into account the current rotation, specified by `vst_rotation`.

This function requires that FSM-GDOS is loaded.

SEE

`vst_rotation`

SYNOPSIS

```
#include <vdi.h>

vqt_cachesize(handle,which,size);

int handle;                workstation handle
int which;                 cache to obtain size of
long *size;                size of selected cache
```

DESCRIPTION

`vqt_cachesize` obtains the size of one of the FSM caches. The size of the cache required is dictated by the `which` parameter; this is 0 to find the size of the largest space in the bitmap cache, or 1 to find the size of the largest space in the data structure cache. The size of the selected cache is returned in `size`.

This function requires that FSM-GDOS is loaded.

SEE

`v_flushcache, v_loadcache, v_savecache`

SYNOPSIS

```
#include <vdi.h>

vqt_devinfo(handle,devnum,exists,name);

int handle;                workstation handle
int devnum;                device number to investigate
short *exists;             non-zero if device exists
char *name;                name of device (if present)
```

DESCRIPTION

`vqt_devinfo` is used to ascertain whether a particular driver ID has been installed and what driver is associated with it. `handle` contains a workstation handle for the device. On return `exists` is non-zero if the device exists, whilst `name` contains the ASCII name for the device.

This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`v_opnwk, v_opnvwk`

EXAMPLE

```
/*
 * Obtain all valid workstation IDs and print out the
 * name of their driver
 */

#include <aes.h>
#include <vdi.h>

#define MAX_SCREEN_ID 10
#define MAX_WORKSTATION_ID 128

int main(void)
{
   short h,exists;
   char name[128];
   int i,dev;
   short work_in[11], work_out[57];
```

```
   appl_init();
   for (dev = 1; dev <= MAX_WORKSTATION_ID; dev++) {
      for (i=1; i<10; i++)
         work_in[i]=1;
      work_in[0] = dev;
      work_in[10] = 2;
      if (dev <= MAX_SCREEN_ID) {
         h=graf_handle(&h,&h,&h,&h);
         v_opnvwk(work_in,&h,work_out);
      }
      else
         v_opnwk(work_in,&h,work_out);
      if (h) {
         vqt_devinfo(h,dev,&exists,name);
         if (exists)
            printf("ID=%d, \"%s\"\n",dev,name);
         if (dev<=MAX_SCREEN_ID)
            v_clsvwk(h);
         else
            v_clswk(h);
      }
   }
   return appl_exit();
}
```

SYNOPSIS

```
#include <vdi.h>

vqt_f_extent(handle,str,pts);

int handle;                 workstation handle
const char *str             string to find size of
short *pts;                 values returned
```

DESCRIPTION

This function returns the screen area needed to display a string of graphics text using the current text attributes, taking *into account* any remainders indicated by `vqt_advance`. This gives how much screen area will be used if `v_ftext` is used to display that string. The diagram below shows how the points that mark the boundary of the string are numbered:

The `pts` array, which should be large enough to hold 8 shorts will be returned as follows:

`pts[0]`	x co-ordinate of point 1.
`pts[1]`	y co-ordinate of point 1.
`pts[2]`	x co-ordinate of point 2.
`pts[3]`	y co-ordinate of point 2.
`pts[4]`	x co-ordinate of point 3.
`pts[5]`	y co-ordinate of point 3.
`pts[6]`	x co-ordinate of point 4.
`pts[7]`	y co-ordinate of point 4.

This function requires that FSM-GDOS is loaded.

SEE

v_ftext, vqt_advance

vqt_f_name — Return font name and index

SYNOPSIS

```
#include <vdi.h>

index=vqt_f_name(handle, num, name, isfsm);

int index;              the font index
int handle;             workstation handle
int num;                font number
char name[32];          font name
short *isfsm;           non-zero if FSM outline font
```

DESCRIPTION

This function returns the name of a font and its font index. The function that changes the current font, `vst_font`, requires a font index which should be obtained using `vqt_f_name`.

The font numbers that are passed in the `num` parameter start at 1 and are followed by 2, 3, etc until the number of loaded fonts. The number of loaded fonts is returned by the `vst_load_fonts` call. Font number 1 is the system font.

The `name` parameter must point to a buffer of at least 32 characters which will be filled in to give the font name.

The `isfsm` flag is set to 1 by the binding to indicate that the selected font is an FSM outline font. If the font is a bitmap font then 0 is returned.

This function requires that Font-GDOS or FSM-GDOS is loaded.

RETURNS

This function returns the font index.

SEE

vqt_name, vqt_f_extent

SYNOPSIS

```
#include <vdi.h>

vqt_get_tables(handle,gascii,style);

int handle;                workstation handle
short **gascii;            pointer to GASCII table
short **style;             pointer to style table
```

DESCRIPTION

`vqt_get_tables` returns the addresses of two tables, which are used internally by FSM-GDOS to translate ASCII characters passed to `v_gtext` into a short word which is recognised by the FSM character generator. These translations are contained in the `gascii` table. This table contains 223 entries with the first entry giving the translation for character 32, the second for character 33....etc.

The second table, `style`, gives information on which font file the character was generated from. When a character is generated the FSM module selects either the main, symbol or Hebrew font based on this table, the values of which are:

Value	Meaning
0	From main font file
1	From symbol font file
2	From Hebrew font file

These tables may then be used by the application to access fonts available in the FSM character sets by modifying the values stored there.

This function requires that FSM-GDOS is loaded.

SEE

`v_gtext`

SYNOPSIS

```
#include <vdi.h>

set = vst_arbpt(handle,point,ch_w,ch_h,cell_w,cell_h);

int set;                point size selected
int handle;             workstation handle
int point;              requested point size
short *ch_w;            character width
short *ch_h;            character height
short *cell_w;          cell width
short *cell_h;          cell height
```

DESCRIPTION

This function selects an arbitrary point size for an FSM font. This differs from the `vst_point` call which will only allow the sizes mentioned in the EXTEND.SYS file to be selected.

This function requires that FSM-GDOS is loaded.

SEE

`vst_point`

SYNOPSIS

```
#include <vdi.h>

vst_error(handle, mode, err);

int handle;              workstation handle
int mode;                error mode
short *err;              pointer to error variable
```

DESCRIPTION

`vst_error` configures the way in which FSM errors are reported. The default, `mode==1`, places FSM error messages on the screen. If `vst_error` is called with `mode==0` then future errors are placed in the `err` variable. The following error codes are used:

Value	Meaning
0	No error
1	Character not found in font
8	Error reading file
9	Error opening file
10	Bad file format
11	Out of memory/cache full
-1	Miscellaneous error

An FSM error may be generated by any of the following calls:

```
v_ftext(), v_gtext(), v_justified(), v_opnvwk(),
v_opnwk(), vqt_advance(), vqt_extent(), vqt_f_extent(),
vqt_f_name(), vqt_fontinfo(), vqt_name(), vqt_width(),
vst_arbpt(), vst_font(), vst_height(), vst_load_fonts(),
vst_point(), vst_setsize(), vst_unload_fonts()
```

This function requires that Font-GDOS or FSM-GDOS is loaded.

SYNOPSIS

```
#include <vdi.h>

vst_scratch(handle, mode);

int handle;                 workstation handle
int mode;                   scratch buffer mode
```

DESCRIPTION

`vst_scratch` sets the manner in which the size of the VDI effects buffer (used by effects such as bold, italic etc.) is calculated. The `mode` parameter should have one of the following values:

Value	Meaning
0	Account for effects on FSM fonts when calculating buffer size
1	Ignore FSM fonts when calculating buffer size
2	Allocate no scratch buffer (hence making any effect use illegal)

This function requires that Font-GDOS or FSM-GDOS is loaded.

SEE

`vst_effects`

SYNOPSIS

```
#include <vdi.h>

set =
vst_setsize(handle,point,ch_w,ch_h,cell_w,cell_h);

int set;                    point size selected
int handle;                 workstation handle
int point;                  requested point size
short *ch_w;                character width
short *ch_h;                character height
short *cell_w;              cell width
short *cell_h;              cell height
```

DESCRIPTION

`vst_setsize` sets the graphic text character width in points. This allows an arbitrary set size to be used for the character width. Note that the next call to `vst_point`, `vst_arbpt` or `vst_height` will override any set size set by this call. This function requires that FSM-GDOS is loaded.

SEE

`vst_arbpt, vst_height, vst_point`

SYNOPSIS

```
#include <vdi.h>

set = vst_skew(handle,skew);

int set;                    skew value selected
int handle;                 workstation handle
int skew                    angle of skew required
```

DESCRIPTION

`vst_skew` sets the skew used when generating characters, note that this is independent of the skewing generated using `vst_effects`. The `skew` value (between -900 and 900) represents the number of 10^{ths} of a degree by which the characters are to be skewed. Negative values produce skews to the left, positive values skews to the right. This function requires that FSM-GDOS is loaded.

RETURNS

The function returns the skew value actually set.

SEE

`vst_effects`

Notes