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.

HiSoft
High Quality Software

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`

wind_set — *Set window attributes*

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).

getcookie — Find cookie in BIOS cookie jar

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;
   }
}
```

putcookie — Put cookie in BIOS cookie jar

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;
   }
}
```

cpx_key — *CPX keyboard event handler*

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()`.

CPX_Save — Save CPX configuration information

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.

MFsave — Save/restore application mouse pointer

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);
```

Popup — Manage CPX popup menus

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;
…
}
```

rsh_fix — Fix-up RCS2 style object tree

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.

Sl_arrow — Implement slider arrows

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;
…
}
```

Sl_size — *Size elevator of scroll control*

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);
```


Xform_do

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`

findfirst, findnext — Find directory entry

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

getdate, setdate — Get/set system date

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