
# GraphApp Reference Manual

Version 3.4

GraphApp Documentation Team
Enchantia
enchantia.com/software/graphapp

# Contents

# Chapter 1

# Introduction

## 1.1 About GraphApp

### What Is GraphApp?

**GraphApp** is a cross-platform graphics library written in the C programming language. It works with a variety of operating system and windowing system combinations, including Microsoft Windows, X-Windows, Linux, Macintosh OSX, and other Unixes. It gives the programmer access to buttons, text boxes, windows, fonts, colours, and so on, in a platform independent and easy to learn manner, so that graphical programs can be easily created and ported to different operating systems.

**About the Author** The author of GraphApp is L. Patrick, who has been employed by the University of Sydney, Australia, as a lecturer and tutor in the Basser Department of Computer Science, in the areas of user interface design, computer graphics and software engineering. GraphApp has been used in teaching courses at the University.

### Use and Distribution of GraphApp

GraphApp is distributed under the **App Software Licence**. A copy of this licence can be found in the file LICENCE.TXT. GraphApp is free software and is distributed in the hope it will be of use, but the software has no warranty.

## 1.2 The GraphApp User Interface

**EMULATED INTERFACE**

GraphApp allows programs to have a graphical user interface. This interface is designed to look very similar to many common graphical user interfaces, such as the Microsoft Windows interface and the various X-Windows interfaces, such as Motif and Gnome.

GraphApp does not use other GUI toolkits or native widget sets to implement its graphical user interface. Instead, it uses its own code. This means that GraphApp can provide its own features, such as support for Unicode fonts and text, which other toolkits may be lacking. Conversely, GraphApp widgets may not work in precisely the same manner as the native widgets on a given platform.

Because GraphApp emulates the widgets found on other platforms, it is a cross-platform programming tool. That is, it is possible to create a GraphApp program and compile it for two different operating systems (for example, Windows and Linux) and it will look and function the same way on both platforms. This is a different approach to portability than is advocated by those who assert that a graphical program's appearance on each platform should blend in with the appearance of the other applications on that platform. Both approaches to portability are valid and have their uses. GraphApp is designed with inter-platform portability in mind, rather than the more common "local look and feel" approach.

**UNICODE SUPPORT**

GraphApp provides support for Unicode fonts and text encodings for internationalised text. Internally, GraphApp code assumes the use of the UTF-8 text encoding. A bitmapped Unicode font is also provided with GraphApp to allow display and editing of UTF-8 encoded Unicode strings.

Text can be cut, copied and pasted using Control-X, Control-C and Control-V respectively. GraphApp checks to see if the text can be represented in one-byte-per-character format, and if so, pastes text in that format. This gives interoperability with many existing applications, which assume text on the clipboard is in that format. GraphApp will accept from the clipboard such text or else UTF-8 text. It converts all text into UTF-8 for use internally.

There is a way to input Unicode characters into any GraphApp program. The ALT key can be used to compose some characters. For example, ALT ' e produces é and ALT / O produces Ø. This composition technique allows the input of many glyphs from European languages.

The Alt key is held down to produce the accent, then released, followed by the letter. The Shift key may also be needed to type the appropriate accent or the following character.

For example:
Alt ˜ n produces the Spanish small letter n with tilde: ñ
Alt / o produces the Danish/Norwegian small letter o with a slash: ø
Alt ‘ E produces capital E with a grave accent: È
Alt A E produces capital AE ligature: Æ

The general rules are:
Alt ’ produces an acute (rising line) above a letter.
Alt ‘ produces a grave (falling line) above a letter.
Alt ˜ produces a tilde above a letter: ñ
Alt ˆ produces a circumflex above a letter: â
Alt “ produces a diaeresis (double dots) above a letter: ö
Alt . produces a dot above a letter.
Alt - produces a macron (line) above a letter, or a stroke through it: Đ
Alt _ produces a horizontal line joined to the bottom of a letter.
Alt | produces a vertical line joined to the left of a letter.
Alt / produces an angled stroke through a letter: Ø
Alt , produces a cedilla for consonants or an ogonek for vowels: ç
Alt u produces a breve (u shape) above a letter.
Alt n produces an inverted breve above a letter.
Alt v produces a caron (v shape) above a letter.
Alt o produces a ring above a letter: Å
Alt O produces a ring around a letter, e.g. copyright or registered symbol.

Ligatures are produced by typing the two letters, so
Alt o e produces lowercase oe ligature (not e with a ring above!)
Alt D z produces Dz ligature.

Alt 1 2 produces the fraction one half: $\frac{1}{2}$. Some other fractions work similarly.

If the second letter typed is a space, it can modify the first.
Alt ! (space) produces upside-down exclamation mark: ¡
Alt ? (space) produces upside-down question mark: ¿
Alt x (space) produces a multiplication symbol.
Alt ’ (space) produces an acute accent (similar for other accents listed).

Greek letters are produced phonetically by preceding with Alt \
Alt \ a produces lowercase alpha
Alt \ p produces lowercase pi
Alt \ f produces lowercase phi
Alt \ u produces lowercase upsilon
Alt \ U produces uppercase upsilon
In some cases there is no obvious equivalent in English:
Alt \ Y produces uppercase psi
Alt \ H produces uppercase eta
Alt \ C produces uppercase xi
Alt \ X produces uppercase chi
Alt \ v produces lowercase non-final sigma

Cyrillic letters are produced phonetically by preceding with Alt ]
Some letters require Alt [ as the prefix instead.
Alt ] A produces uppercase Cyrillic A.
Alt ] T produces uppercase Cyrillic TE.
Alt [ T produces uppercase Cyrillic TSE.

Any GraphApp program which has a text entry field can thus be used to compose European characters, for pasting into other applications.

**PORTABILITY SUPPORT**

GraphApp implements a core set of routines which behave the same way on each platform. This core set include drawing routines, font selection, and navigation of folders (directories).

Folder names are assumed to be separated by forward slashes, following the Web and Unix convention. So, a program would open a file using `open_file("/My Documents/stuff.txt", "r")` rather than `fopen("\My Documents\stuff.txt, "r")` on a Windows platform (or any platform, for that matter). Internally, GraphApp converts forward slashes to whatever the native directory separator is. This allows GraphApp programs a great degree of portability. Folder and file names can be represented in one format in source code and in data files, and the library handles the conversion to whatever the operating system requires.

Text can be written to and read from files in either the UTF-8 or ISO-Latin-1 encodings. All text is converted to the UTF-8 format internally, so it is not valid to assume one character is stored in one byte if you use the GraphApp file input

routines. This is convenient for manipulating Unicode text, such as a document which contains a mixture of English, Greek and Chinese text.

If you read data into a GraphApp program as ISO-Latin-1 text (one byte per character) it will probably be displayed and edited correctly, but there is no guarantee that this will always work. For example, the ligature Æ followed immediately by the English pound sign £ is actually a UTF-8 encoding for a completely different letter (Latin small letter OI). To avoid this problem, use the GraphApp input routines to correctly convert the data based on its original encoding.

Because GraphApp implements drawing routines portably, the appearance of a program will generally be exactly the same on another computer or operating system. This means you can rely on the pixel placement of buttons, lines, and fonts (if using the portable fonts included in the GraphApp package).

One problem with this pixel-level approach to portability is that screen resolutions may differ, and so the bitmapped font supplied with GraphApp may not be of a good resolution for all platforms. In practice, this seems to not be too much of a problem, because the font is quite large (it has to be, to allow correct rendering of complex Asian glyphs). Also, GraphApp programs can be configured to use native fonts, such as Helvetica and Times, although this usually precludes the correct display of Unicode glyphs.

**RESOURCES**

Application resources can be added into a GraphApp application. A resource is a font or other file or directory structure containing files. These can be added on to the end of a compiled program, and used at run-time by that program. The tools directory within the GraphApp package contains some tools for manipulating resources, as well as documentation.

One benefit of resources is that the portable Unicode font can be added onto a compiled application, and that program can then be distributed to friends or clients, and there is no installation required for them to use the program. The font will automatically be found and used by the GraphApp program because it is visible to GraphApp within the program's resources.

This means it is possible to completely guarantee a program's look-and-feel using GraphApp, down to the availability of a Unicode font and its pixel representation. This level of portability is not even available within Java.

Resources are simply files tacked on the end of a program's executable file, separated by a single NUL byte and indexed by a "resource directory" data structure which is the last thing added to the application. It's quite a simple but powerful

idea, because it's trivial to implement, but saves a lot of tedious mucking about with installation programs.

# Chapter 2

# General Concepts

## 2.1 Initialisation and Event Handling

**HEADERS**

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

**OBJECTS**

```
typedef struct App  App;

struct App {
  int       gui_available;    /* is the GUI available? */
  Rect      screen_area;      /* screen pixel dimensions */
  Rect      screen_mm;        /* size in millimetres */
  int       num_windows;      /* a list of windows */
  Window ** windows;
  int       visible_windows;  /* how many windows visible */
  int       fonts_loaded;     /* a list of loaded fonts */
  Font **   fonts;
  char *    program_name;     /* absolute path to program */
  int       has_resources;    /* is it a resource file? */
  void *    data;             /* user-defined data */
  void *    extra;            /* platform-specific data */
};
```

**FUNCTIONS**

```
int   main(int argc, char **argv);        /* definition of main */

App * new_app(int argc, char **argv); /* initialise library */
void  del_app(App *app);              /* shut-down program */

void  main_loop(App *app);            /* handle all events */
int   peek_event(App *app);           /* is there an event? */
int   wait_event(App *app);           /* wait for one event */
int   do_event(App *app);             /* handle one event */

int   exec(App *app, char *cmd);      /* execute a program */

void  error(App *app, char *message); /* exit with message */
```

**NOTES**

The two header files which must be included in any GraphApp program are named `<stdio.h>` and `<graphapp.h>`, and they must be included in that order, since the GraphApp functions make use of the `stdio` *FILE* type. Programs begin, as usual, in the **main** function, which *must* be defined as above.

The function **new_app** initialises the structures and variables necessary to use the graphics library's interface, storing the data into a structure called an *App*, and returning a pointer to it. This pointer is required by other functions. If there is no memory remaining, NULL will be returned. If the program is being run from a non-graphical terminal, the structure will still be created, the `gui_available` field will be set to zero, and some functions will still work, such as drawing into images, but functions which require access to the windowing system, such as creating a window, or drawing to a bitmap, will not work.

If the graphical user interface could be initialised, the `gui_available` field will be set to non-zero. The `screen_area` field will be initialised to the screen's dimensions in pixels. Typically the `x` and `y` fields of this rectangle will be zero, while the `width` and `height` fields will contain the size of the screen in pixels. The `screen_mm` field will contain the dimensions of the screen in millimetres (there are 25.4 millimetres in an inch).

The function takes parameters `argc` and `argv` from the **main** function and searches them for interface-specific options (such as those found in X-Windows environments). If it finds any such initialisation parameters, it removes them from the `argv` list by moving the next parameters forwards in the list, and `argc` will

be invalid after this process (a NULL value in the `argv` array now signals the end of parameters).

The **del_app** function is called at the end of the program. It closes all windows and releases from the memory the *App* structure.

The **main_loop** function is called after creation of graphical objects, such as windows and controls. It polls the windowing system for events and dispatches them to the appropriate callback functions. The function will terminate when there are no windows visible. It is thus possible to call this function again if other windows become visible.

The **peek_event** is true when there are events to be processed in the event queue. It is rarely used, since it polls the windowing system each time.

The **do_event** function checks if there is an event to be processed, and if there is, dispatches it. It returns zero only if there are no more events possible, for instance, if there are no visible windows. It is generally not used, since it polls the windowing system each time it is called.

The **wait_event** function waits until an event is available, then handles that event using **do_event**. If there are no more events possible, it returns zero. It is called repeatedly by the **main_loop** function to handle events.

The **exec** function launches the specified application, returning 1 on success and 0 on failure. This function exists because the standard C function **system** is not always implemented on all platforms (e.g. MS-Windows).

The **error** function displays an error message in a window and then stops the application.

**EXAMPLES**

Here is an example program skeleton:

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

int main(int argc, char *argv[])
{
  App *app;

  /* some initialisation code here */
  app = new_app(argc, argv);
  if ((app == NULL) || (app->gui_available == 0))
```

```
    error(app, "Couldn't initialise the graphics library.");
  /* rest of initialisation and drawing code here */
  main_loop(app);
  del_app(app);
  return 0;
}
```

# Chapter 3

# Simple Objects

## 3.1 Bytes

### OBJECTS

```
typedef unsigned char  byte;
```

### NOTES

A *byte* is simply another name for an *unsigned char*, which is defined by the C language to be an 8-bit integer.

The *byte* type has been defined in this library as a convenient short-hand for an unsigned 8-bit integer, and is used in a few functions. It is quite acceptable to use *unsigned char* instead, if you prefer, since the two are equivalent.

The possible range of numbers which can be stored in a *byte* are 0 to 255 inclusive.

If your compiler does not allow the definition of this type, replace all instances with *unsigned char* instead.

## 3.2 Points

### OBJECTS

```
typedef struct Point  Point;

struct Point {
  int x;    /* horizontal co-ordinate */
  int y;    /* vertical co-ordinate */
};
```

### FUNCTIONS

```
Point pt(int x, int y);
Point new_point(int x, int y);

int   points_equal(Point p1, Point p2);
int   point_in_rect(Point p, Rect r);
```

### NOTES

A *Point* refers to a location in a drawing, which is a bitmap, window or control, using x and y coordinates.

The coordinate system has x increasing to the right and y increasing down. The top-left point of a drawing is always the point (0,0). The pixel corresponding to a point is below and to the right of its coordinates.

Important note: Points, when passed as function parameters, are generally passed by value on the stack. This means that modifying a point within a function will not change its co-ordinates outside that function. Points can thus be treated as numeric objects, like integers. (Were they to be passed by pointer, this would not be the case.) This differs somewhat from the way Java implicitly passes all objects by pointer, except for numbers.

To create a new point, call the function **pt**(x,y). This is a macro which actually calls **new_point**, but has a shorter name, for convenience.

The **points_equal** function compares two points and returns non-zero if they are equal, zero if they are not.

The **point_in_rect** function returns non-zero if the given point is within the given rectangle, zero otherwise.

## 3.3 Rectangles

### OBJECTS

```
typedef struct Rect  Rect;

struct Rect {
  int x, y;           /* top-left point inside rectangle */
  int width, height;  /* width and height in pixels */
};
```

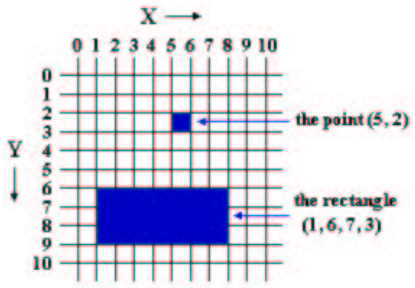
### FUNCTIONS

```
Rect  rect(int left, int top, int width, int height);
Rect  new_rect(int left, int top, int width, int height);

Rect  inset_rect(Rect r, int i);      /* by i pixels */
Rect  center_rect(Rect r1, Rect r2);  /* center r1 on r2 */
int   rects_equal(Rect r1, Rect r2);  /* is r1 equal to r2? */
int   rect_in_rect(Rect r1, Rect r2); /* is r1 inside r2? */
int   rect_intersects_rect(Rect r1, Rect r2);
Rect  clip_rect(Rect r1, Rect r2);    /* clip r1 inside r2 */
Rect  rect_abs(Rect r);               /* force positive size */
```

### NOTES

A *Rect* defines a rectangular area. The x and y co-ordinates define the top-left point within the rectangle, and the rectangle's width and height are recorded in pixels. The point (x+width,y+height) will thus be outside the rectangle.




Important note: Rectangles, when passed as function parameters, are generally passed by value on the stack. This means that modifying a rectangle within a

function will not change its co-ordinates outside that function. Rectangles can thus be treated as numeric objects, like integers. (Were they to be passed by pointer, this would not be the case.) This differs somewhat from the way Java implicitly passes all objects by pointer, except for numbers.

A new rectangle can be returned using **rect**(x,y,width,height). This is a macro which just calls **new_rect**, but which has a shorter name for convenience.

The **inset_rect** function returns a rectangle which is inset from the given rectangle by the specified number of pixels all the way around. So **inset_rect**(r,i) is equivalent to **rect**(r.x+i,r.y+i,r.width-i-i,r.height-i-i).

A rectangle can be centered within another rectangle using the **center_rect** function: **center_rect**(r1,r2) will return a rectangle with the same size as r1, but centered within r2, even if r2 is smaller than r1.

The **rects_equal** function compares two rectangles, returning non-zero if they are equal, zero otherwise.

Calling **rect_in_rect**(r1, r2) returns non-zero only if r1 is wholly contained within r2. By contrast, **rect_intersects_rect**(r1, r2) returns non-zero if any part of r1 intersects with r2.

The **clip_rect** function clips a rectangle so that it is within another: **clip_rect**(r1, r2) will clip r1 to be within r2 and return r1, unless r1 does not overlap r2 at all, in which case it will return r1 unchanged.

The **rect_abs** function converts the supplied rectangle to canonical form; the width and height of the returned rectangle will be positive, and the area and location of the rectangle will remain unchanged.

## 3.4 Regions

### OBJECTS

```
typedef struct Region  Region;

struct Region {
  Rect    extents;    /* enclosing rectangle */
  int     size;       /* allocated size of rectangle list */
  int     num_rects;  /* list of non-intersecting rects */
  Rect *  rects;
};
```

### FUNCTIONS

```
Region * new_region(void);
void     del_region(Region *rgn);
Region * new_rect_region(Rect r);
Region * copy_region(Region *rgn);

void move_region(Region *rgn, int dx, int dy);
int  union_region(Region *rgn1, Region *rgn2, Region *dest);
int  union_region_with_rect(Region *rgn, Rect r, Region *dest);
int  intersect_region(Region *rgn1, Region *rgn2, Region *dest);
int  intersect_region_with_rect(Region *rgn, Rect r, Region *dest);
int  subtract_region(Region *rgn1, Region *rgn2, Region *dest);
int  xor_region(Region *rgn1, Region *rgn2, Region *dest);

int  region_is_empty(Region *rgn);
int  regions_equal(Region *rgn1, Region *rgn2);
int  point_in_region(Point p, Region *rgn);
int  rect_intersects_region(Rect r, Region *rgn);
int  rect_in_region(Rect r, Region *rgn);
```

### NOTES

A *Region* defines an arbitrary collection of points. It is implemented as an array of non-intersecting rectangles.

Important note: Regions, unlike *Rect* objects, are usually passed by pointer, not on the stack. Hence, changing a region within a function will usually also change it outside that function.

A new empty region can be returned using **new_region**. The function returns NULL if it runs out of memory and cannot create the region.

The **del_region** function should be used to delete regions when they are no longer needed.

The **new_rect_region** function creates a new region which contains all points within a given rectangle. It returns NULL if it runs out of memory.

To create a complete copy of an existing region, use **copy_region**. This function returns NULL if it runs out of memory.

To move a region to a new location, **move_region** is used. It adds `(dx,dy)` to all points within a region.

**union_region** forms in `dest` a region which is the union of all points within the regions `rgn1` and `rgn2`. The `dest` parameter must be an existing region, which may or may not be empty. The function returns one on success, zero if it runs out of memory. The union of two regions is defined as all points which are within either region.

**union_region_with_rect** performs the same operation, but using a rectangle as one of the items in the union.

**intersect_region** forms the intersection of two regions, placing the result in `dest`, which must be an existing, possibly empty, region. It returns one on success, zero if it runs out of memory. The intersection of two regions is defined as all points within both regions.

**intersect_region_with_rect** performs the same operation, but using a rectangle as one of the items to be intersected.

**subtract_region** forms a region which is `rgn1` with `rgn2` removed, storing the result into the already existing, possibly empty, region `dest`. Only `dest` is changed. This will be the set of all points which are inside `rgn1`, but not inside `rgn2`. The function returns one on success, zero if it runs out of memory.

**xor_region** forms the exclusive-or of two regions, placing the result in `dest`, which must be an existing, possibly empty, region. It returns one on success, zero if it runs out of memory. The exclusive-or of two regions is defined as all points which are within either region `rgn1` or `rgn2`, but not within both regions.

**region_is_empty** return one if the region is empty, zero if it is not. A region is empty if it encloses no points.

**regions_equal** returns one if the two regions enclose the same set of points, or

zero if they differ.

**point_in_region** returns one if the given point is within the region, or zero if it is not.

**rect_intersects_region** returns one if any part of the given rectangle lies within the region (some parts of the rectangle may also lie outside the region). It returns zero if they are completely distinct.

**rect_in_region** returns one if the *entire* rectangle lies within the region, or zero if any part of the rectangle lies outside the region.

## 3.5 Colours

**OBJECTS**

```
  typedef struct Colour  Color;
  typedef struct Colour  Colour;

  struct Colour {
    byte  alpha;  /* 0=opaque, 255=transparent */
    byte  red;    /* intensity of red light, 0=none, 255=bright */
    byte  green;  /* intensity of green light, 0=none, 255=bright */
    byte  blue;   /* intensity of blue light, 0=none, 255=bright */
  };
```

**FUNCTIONS**

```
  Colour  rgb(int red, int green, int blue);     /* create Colour */
  Colour  argb(int alpha, int r, int g, int b);  /* with al-
pha */

  Colour  new_color(int r, int g, int b);    /* create Colour */
  Colour  new_colour(int r, int g, int b);   /* create Colour */

  int     rgbs_equal(Colour a, Colour b);    /* compare Colours */
  int     colors_equal(Colour a, Colour b);  /* compare Colours */
  int     colours_equal(Colour a, Colour b); /* compare Colours */
```

**CONSTANTS**

```
  #define CLEAR           argb(0xFF,0xFF,0xFF,0xFF)

  #define BLACK           rgb(0x00,0x00,0x00)
  #define WHITE           rgb(0xFF,0xFF,0xFF)
  #define BLUE            rgb(0x00,0x00,0xFF)
  #define YELLOW          rgb(0xFF,0xFF,0x00)
  #define GREEN           rgb(0x00,0xFF,0x00)
  #define MAGENTA         rgb(0xFF,0x00,0xFF)
  #define RED             rgb(0xFF,0x00,0x00)
  #define CYAN            rgb(0x00,0xFF,0xFF)
```

```
#define GREY            rgb(0x80,0x80,0x80)
#define GRAY            rgb(0x80,0x80,0x80)
#define LIGHT_GREY      rgb(0xC0,0xC0,0xC0)
#define LIGHT_GRAY      rgb(0xC0,0xC0,0xC0)
#define DARK_GREY       rgb(0x40,0x40,0x40)
#define DARK_GRAY       rgb(0x40,0x40,0x40)

#define DARK_BLUE       rgb(0x00,0x00,0x80)
#define DARK_GREEN      rgb(0x00,0x80,0x00)
#define DARK_RED        rgb(0x80,0x00,0x00)
#define LIGHT_BLUE      rgb(0x80,0xC0,0xFF)
#define LIGHT_GREEN     rgb(0x80,0xFF,0x80)
#define LIGHT_RED       rgb(0xFF,0xC0,0xFF)
#define PINK            rgb(0xFF,0xAF,0xAF)
#define BROWN           rgb(0x60,0x30,0x00)
#define ORANGE          rgb(0xFF,0x80,0x00)
#define PURPLE          rgb(0xC0,0x00,0xFF)
#define LIME            rgb(0x80,0xFF,0x00)
```

**NOTES**

A *Colour* value is a structure used to represent a colour, using four integer components: three to represent the intensity of red, green and blue light, and one to represent the transparency of the colour against a background, otherwise known as the alpha channel value. The names *Colour* and *Color* are interchangeable.

The **rgb** function constructs a fully-opaque *Colour* value, given the required intensity of red, green and blue light. The values of red, green and blue can range from zero (no intensity) to 255 (maximum intensity). The **rgb** function is implemented as a macro.

Pre-defined colours are available, such as **BLACK, WHITE, GREY, BLUE, RED, GREEN,** etc. These are defined using the C pre-processor as calls to the **rgb** function, and as such, cannot be used outside of function blocks (for instance, you cannot use the pre-defined colours in an initialising structure). The bit-patterns these colours become when inside a bitmap are not generally defined; bitmaps are platform-dependent data structures, and should be treated as output devices only. (Note that both American and British English spellings are supported, both for colour-names and functions which employ the word 'colour' or 'color'. Thus **GREY** and **GRAY** are quite legal, and interchangeable.)

The alpha byte of a *Colour* value is used to represent a transparency value. If this byte is equal to 255, that Colour is fully transparent. If the alpha value is zero, the Colour is fully opaque. In theory, an alpha value between zero and 255 can be

used to perform 'colour blending', however currently only fully transparent and fully opaque rgb values are supported. The special constant **CLEAR** can be used to represent fully transparent Colour values in images, and so forth.

The **argb** function constructs a *Colour* value which has an explicit alpha value between 0 and 255 inclusive.

The **new_color** and **new_colour** functions are macros which call **rgb** to create fully-opaque colours.

Colours can be be compared using **rgbs_equal**. The macro functions **colors_equal** and **colours_equal** are synonyms for this comparison function.

Important note: Colours, when passed as function parameters, are generally passed by value on the stack. This means that modifying a colour within a function will not change its value outside that function. Colours can thus be treated as numeric objects, like integers. (Were they to be passed by pointer, this would not be the case.) This differs somewhat from the way Java implicitly passes all objects by pointer, except for numbers.

## 3.6 Palettes

### OBJECTS

```
typedef struct Palette  Palette;

struct Palette {
  int     size;       /* number of colours */
  Colour *element;    /* array of colours */
};
```

### FUNCTIONS

```
Palette * new_palette(int size, Colour *elem);
void      del_palette(Palette *pal);

void      set_window_palette(Window *w, Palette *pal);
Palette * get_window_palette(Window *w);

byte *    palette_translation(Palette *target, byte *dest,
                              int size, Colour *elem);
```

### NOTES

A *Palette* defines an indexed array of up to 256 colours. The colours are usually all different, but do not have to be.

The **new_palette** function creates a palette by copying the supplied array of colours, given the number of elements to copy and a pointer to the start of the array. Because the palette is a copy of the colours, the supplied array may be deleted after this function completes.

A palette should be destroyed using **del_palette**.

The **set_window_palette** functions sets the window's private palette to a copy of the given set of colours, if the window manager supports this operation. If the depth of the screen is greater than 8 (for example, if the screen is in TrueColour or DirectColour mode) this operation will do nothing, since there are already more colours available than would be possible using a private window palette. If the screen cannot display enough colours to display the entire requested palette, the first N entries will be used, where N is the number of colours the screen can display. Therefore, the palette should be sorted in order of most important colours,

for maximum portability. Passing a NULL palette pointer to this function removes any private palette from the window.

If a window has a private palette, the window manager guarantees to display that set of colours if the window has mouse focus. When another window has focus, the colours may be remapped. Usually all windows share the common system palette, so that no colour 'flashing' occurs when focus changes between windows. Such 'flashing' is minimized in this library due to the way **set_window_palette** matches the requested palette to the closest set of colours in the system palette *before* the window manager is notified of the request.

The **get_window_palette** function returns a pointer to the actual palette used by a window. This will be a copy of the requested palette, but might be smaller if not all the colours could be allocated. It returns NULL if the window does not have private palette, or if a private palette is not needed (for instance, if the screen is in TrueColour mode).

The **palette_translation** function writes into `dest` a series of integers which show the best match between the colours in the array of colours named `elem` and the `target` palette, such that `elem[i]` is closest to `palette->element[dest[i]]` for `0<=i<size`. It returns `dest`.

This function can be used to determine which entry in a palette has the closest visual match to one given colour (if `size` is 1, and `elem` points to one colour) or it can generate an entire list of matches at once, for efficiency. The matching approximates the way the human eye matches colours. `dest` must be an array large enough to hold all the answers; it must be at least `size` bytes long. Note that although a palette can only hold 256 colours, the `elem` array might hold more if, for example, it is a scanline of colours within an Image that is being matched to a palette.

## 3.7 Fonts

**OBJECTS**

```
typedef struct Font        Font;
typedef struct Subfont     Subfont;
typedef struct FontWidth   FontWidth;

struct Font {
  int             refcount;       /* used when caching fonts */
  int             maximum_width;  /* maximum character width */
  int             height;         /* pixel height of chars */
  char *          name;           /* UTF-8 encoded font name */
  int             style;          /* style */
  App *           app;            /* back pointer to cache */
  int             num_subfonts;
  Subfont **      subfonts;       /* list of subfonts */
  void *          extra;          /* platform-specific data */
};

struct Subfont {
  unsigned long   base;           /* Unicode value of first char */
  int             num_widths;     /* list of font width records */
  FontWidth **    widths;
  void *          extra;          /* platform-specific data */
};

struct FontWidth {
  int             width;          /* in pixels, -1=non-existent */
  int             num_ranges;     /* number of (start,end) pairs */
  byte *          range_list;     /* (start,end) pairs of bytes */
};
```

**FUNCTIONS**

```
Font *new_font(App *app, char *name, int style, int height);
void  del_font(Font *f);

int   font_height(Font *f);
int   font_width(Font *f, char *utf8, int nbytes);

Font *find_default_font(App *app);
```

```
void  set_font(Graphics *g, Font *f);
void  set_default_font(Graphics *g);
```

**CONSTANTS**

```
enum {
  PLAIN  = 0,
  BOLD   = 1,
  ITALIC = 2,
  PORTABLE_FONT = 16,
  NATIVE_FONT   = 32
};
```

**NOTES**

A *Font* is a typeface of a certain size and style.

A font can be obtained using **new_font**. The name parameter is a UTF-8 encoded string like “serif” or “unifont”, while the style is a bit-field composed of any of the styles **PLAIN**, **BOLD**, **ITALIC**, **PORTABLE_FONT** or **NATIVE_FONT**. Use the bitwise-or operator to combine these styles. The height field specifies a pixel-height for the font. The function may return NULL if it cannot find a suitable font.

Hence **new_font**(app, “serif”, BOLD, 12) returns a bold serif 12-pixel-high font.

If the **PORTABLE_FONT** style flag is included, only portable fonts supplied with the library will be searched for a matching font. If the **NATIVE_FONT** style flags is given, only native platform-specific fonts will be searched. Native fonts are inherently non-portable and usually do not handle anything more than ISO-Latin-1 text, but are generally rendered fast. If neither of these flags, or both of them, are given, first portable fonts and then native fonts are searched for a matching name, style and size; the first matching font is returned.

The **del_font** function releases the memory used by a font. In general this function does not need to be used unless memory is critical, as fonts will automatically be released from memory when the program ends.

The **font_height** functions reports the pixel height of the font. All characters within the font will use this height as their inter-line spacing.

The **font_width** function reports the pixel width of the given UTF-8 encoded string in the supplied font. The `nbytes` parameter specifies the number of bytes within the string, thus allowing ’\0’ characters to be within the string.

The **find_default_font** function returns the default font. The supplied App object's font list is searched for the font first, and loaded into that object if it hasn't already been loaded.

The current font used for drawing within a given *Graphics* context can be set using the **set_font** function.

The **set_default_font** function loads the default font and sets that font in the Graphics context. It is the equivalent of `set_font(g, find_default_font(g->app));`

Three portable fonts are currently supplied with GraphApp: "serif", "plain" and "unifont", the last being the default Unicode font. Portable fonts are stored as bitmaps. Each bitmap contains space for exactly 256 characters, and is called a *Subfont*. Subfonts are loaded on demand by the font rendering engine, and cached in a list on the *Font* structure. The details of this caching mechanism, and the other data associated with a subfont, are not, in general, relevant to the programmer.

If a particular character glyph cannot be found on a font by the font rendering engine, it will then search for it on the default font. If it still isn't found, a rectangular box shape will be drawn instead. Since the supplied Unicode font contains some 35,000 characters, this event should be rare in normal usage.

## 3.8 Bitmaps

### OBJECTS

```
typedef struct Bitmap  Bitmap;

struct Bitmap {
  Window *        win;
  Rect            area;
  void *          extra;
};
```

### FUNCTIONS

```
Bitmap * new_bitmap(Window *win, int width, int height);
void     del_bitmap(Bitmap *b);
Rect     get_bitmap_area(Bitmap *b);

Bitmap * image_to_bitmap(Window *win, Image *img);
```

### NOTES

A *Bitmap* holds a rectangular pixel-image in offscreen memory. Bitmaps are very fast to copy to a window or to another bitmap, and are used to store prepared images for quick display. A bitmap can be drawn into, and also used as a source of pixels. Bitmaps can have only one level of transparency, so that pixels are either fully opaque or fully transparent within the bitmap. Drawing into a bitmap renders modified pixels opaque.

The **new_bitmap** function creates and returns an initially transparent bitmap with the specified pixel width and height. The arrangement of data within the bitmap is guaranteed to be the same as the supplied window, so that pixels may be copied from the bitmap to that window and they will represent the same colours. A bitmap always has a zero origin in its top-left corner. The function returns NULL if it runs out of memory.

The **del_bitmap** function releases the memory used by a bitmap.

The **get_bitmap_area** function returns a rectangle containing the dimensions of the bitmap. The top and left co-ordinates of this rectangle will always both be zero.

The **image_to_bitmap** function creates a new bitmap which contains the data from the image. It uses the window's colour data arrangement when deciding how to map the image's colours onto pixel values in the bitmap. This function is efficient, but does no dithering. Opaque pixels in the source image overwrite pixels in the bitmap, and cause those pixels to become opaque. Transparent pixels in the source image are not copied into the bitmap; hence, portions of the produced bitmap may be transparent.

## 3.9 Images

**OBJECTS**

```
typedef struct Image  Image;

struct Image {
  int       depth;      /* depth will be 8 or 32 */
  int       width;      /* image width in pixels */
  int       height;     /* image height in pixels */
  int       cmap_size;  /* size of colour map, may be zero */
  Colour *  cmap;       /* indexed colour map, may be null */
  byte **   data8;      /* array of scanlines, indexed */
  Colour ** data32;     /* array of scanlines, direct colour */
};
```

**FUNCTIONS**

```
Image * new_image(int width, int height, int depth);
void    del_image(Image *img);

Rect    get_image_area(Image *img);

Image * copy_image(Image *img);
void      set_image_cmap(Image *img, int cmap_size, Colour *cmap);

Image * image_convert_32_to_8(Image *img);
Image * image_convert_8_to_32(Image *img);
void    image_sort_palette(Image *img);
Image * scale_image(Image *src, Rect dr, Rect sr);

void  draw_image(Graphics *g, Rect dr, Image *i, Rect sr);
void  draw_image_monochrome(Graphics *g, Rect dr, Image *i, Rect sr);
void  draw_image_greyscale(Graphics *g, Rect dr, Image *i, Rect sr);
void  draw_image_darker(Graphics *g, Rect dr, Image *i, Rect sr);
void  draw_image_brighter(Graphics *g, Rect dr, Image *i, Rect sr);

Bitmap *image_to_bitmap(Window *win, Image *img);
```

**NOTES**

An *Image* is a platform-independent representation of a picture. It is a more general representation than a bitmap, but a bitmap is much faster to copy onto the screen. For this reason, images and bitmaps are often used in conjunction to produce good effects.

The **new_image** function allocates memory for an image of the given width, height and depth. The depth parameter can only be 8 or 32. All other image depths are deliberately not supported. The function returns NULL if there is insufficient memory to create the array of pixel scanlines.

An image of depth 8 uses the cmap array to store a colour palette. The data8 pointer will be an array of *height* lines, each line being an array of *width* bytes, each byte an index into the colour palette. By contrast, an image of depth 32 will have no cmap table, cmap_size will be zero and the data32 array will be an array of *height* lines, each line being an array of *width* colour values.

An image can store transparency information, either using a CLEAR entry in the colour palette of an 8-bit image, or by having CLEAR pixels in a 32-bit image.

The **del_image** function deallocates an image from memory. Images can occupy a lot of memory, particularly if they have a large area or are in 32-bit direct-colour format.

The **get_image_area** function returns a rectangle representing the size of an image. The x and y co-ordinates will contain zero, and the width and height co-ordinates will contain the width and height of the image.

The **copy_image** function will make a completely separate copy of an image, or return NULL if there is insufficient memory.

The **set_image_cmap** function releases from memory any existing colour map on the image, and creates a new one filled with the supplied data. This essentially changes the meaning of the existing pixel values in the data8 array.

The **image_convert_32_to_8** function creates a new 8-bit image from a 32-bit image, or returns NULL if there is insufficient memory. It tries a fast conversion algorithm first (which assumes there are less than 256 colours in the 32-bit image), and resorts to a slower algorithm if that doesn't work.

The **image_convert_8_to_32** function creates a new 32-bit image from an 8-bit image, or returns NULL if there is not enough memory. It uses a very fast, simple algorithm.

The **image_sort_palette** function modifies an existing 8-bit image to optimise its colour palette so that redundant entries are discarded, and the most common

colours are placed at the top of the palette. The data8 pixel values will be modified so they represent the same colours. This may also change the cmap_size and cmap fields in the image. This function has no effect on 32-bit images.

The **scale_image** function produces a new image which is cropped and/or scaled. The pixels from the source image which correspond to the source rectangle **sr** are scaled to fit the destination rectangle **dr**. The resultant image will have the same width and height as the destination rectangle. The x and y values from the destination rectangle are ignored. If the source rectangle lies partially or wholly outside the image's rectangle, the corresponding pixels in the new image will be CLEAR, unless the image has a palette and that palette has no transparent entries, in which case those pixels will have the value zero.

Use **draw_image** to draw an image into the given rectangle to the destination specified by the graphics object. If the destination rectangle is smaller or larger than the source rectangle, the source pixels will be scaled to fit.

The **draw_image_monochrome** function draws an image so that it appears black and white.

The **draw_image_greyscale** function draws an image in five levels of grey (a synonym for this function is **draw_image_grayscale**). This can be used to provide a 'disabled button' effect.

By contrast, **draw_image_darker** draws an image so that it looks darkened, as if seen through dark glass. This can be used to provided a 'clicked button' effect. The **draw_image_brighter** function draws a lighter image.

The **image_to_bitmap** function copies the pixels from an image into a new bitmap. The bitmap will have the same arrangement of colour data as the supplied window, so that copying the bitmap to that window will produce a picture with very similar colours to the original image. Bitmaps may store one level of transparency, so transparent pixels in the image will be transparent in the bitmap.

## 3.10 Image Lists

### OBJECTS

```
typedef struct ImageList  ImageList;

struct ImageList {
  int             num_images;         /* list of images */
  Image **        images;
};
```

### FUNCTIONS

```
ImageList * new_image_list(void);
void  del_image_list(ImageList *imglist);
void  append_to_image_list(ImageList *imglist, Image *img);
```

### NOTES

An *ImageList* is simply a dynamically allocated array of image. It grows as *Images* are appended to it.

The **new_image_list** function allocates memory for an empty image list. The function returns NULL if there is insufficient memory to create the structure.

The **del_image_list** function is used to deallocate the image list structure and all the images which have been added to the list. Therefore, images should only be added to the list if they will not be deleted elsewhere.

The **append_to_image_list** function appends an image to the list, after first reallocating the list to be large enough.

## 3.11 Image Readers

**OBJECTS**

```
typedef struct ImageReader  ImageReader;

typedef int (*ImageMessageFunc) (ImageReader *reader, char *message);
typedef int (*ImageProgressFunc)(ImageReader *reader);

struct ImageReader {
  char *              filename;           /* user-given fields */
  FILE *              file;               /* file to read from */
  Palette *           src_pal;            /* dither to this palette */
  int                 max_cmap_size;      /* only use this many colours */
  int                 required_depth;     /* 8 or 32 */

  ImageMessageFunc    message_func;       /* report warnings */
  ImageProgressFunc   error_func;         /* if error, tidy up */
  ImageProgressFunc   startup_func;       /* called before start */
  ImageProgressFunc   after_dither_func;  /* called after dither */
  ImageProgressFunc   progress_func;      /* called after each line */
  ImageProgressFunc   rendering_func;     /* called after each line */
  ImageProgressFunc   success_func;       /* called after success */
  void *              user_data;          /* user-defined data */

  int                 state;          /* DITHERING, RENDERING etc */
  int                 stage;          /* 1 <= stage <= max_stages */
  int                 max_stages;     /* total number of stages */
  int                 row;            /* 0 <= row < height, random */
  int                 rows_done;      /* 0,1,...,height in order */
  int                 row_height;     /* of current stage */
  int                 width;          /* in pixels */
  int                 height;         /* in pixels */
  byte **             data8;          /* implies palette */
  Colour **           data32;         /* implies no palette */
  Palette *           pal;            /* if data8 used */
};
```

**FUNCTIONS**

```
ImageReader * new_image_reader(void);
void    del_image_reader(ImageReader *reader);
```

```
Image * read_image(char *filename, int required_depth);
Image * read_image_file(FILE *file, int required_depth);
Image * read_image_progressively(ImageReader *reader);
int     find_image_format(FILE *file);
```

**CONSTANTS**

```
enum ImageReaderState {
  STOPPED         = 0,    /* at start or end of processing */
  STARTING,               /* creating data structures */
  DITHERING,              /* now dithering to a palette */
  RENDERING,              /* processing lines of pixels */
  IMAGE_ERROR     = -1    /* an error has happened */
};

enum ImageFormat {
  PNG_FORMAT      = 1,    /* Portable Network Graphics */
  JPEG_FORMAT     = 2,    /* Joint Photographic Experts Group */
  GIF_FORMAT      = 3,    /* Graphic Interchange Format */
  UNKNOWN_FORMAT  = -1
};
```

**NOTES**

An *ImageReader* is an abstract object used to read an *Image* from a file. It can be used to control how an image is read and to receive notifications as lines of the image are completed.

An empty *ImageReader* structure is first obtained using **new_image_reader**, and then fields are assigned to control the reading process. The fields `filename`, `file`, `src_pal`, `max_cmap_size`, and `required_depth` should be assigned values. If dithering to a user-specified palette is not required, the `src_pal` and `max_cmap_size` can be left as their initial empty values.

Call-back functions can be assigned which will notify the program when the image reading process has completed certain stages:

- The **message_func** call-back is used whenever a warning or error message must be issued to the user. Setting this to NULL prevents any messages being shown. Typically this function will display the message in a dialog box, or to the terminal using **printf**.

- The **error_func** call-back is used if a critical error occurs, such as the image being damaged or truncated. Typically this will tidy up and place the program in a state where it is no longer expecting a valid image pointer.
- The **startup_func** call-back is called before any image processing occurs. It can be used to initialise program values needed to display the progress of the image reader. The image `width` and `height` fields will be correct at this point.
- The **after_dither_func** call-back happens after the image reader has dithered the image to the programmer-specified palette (if any). This happens before reading any lines of pixels. The image palette will be correct at this point, and will either be a copy of the programmer-specified palette (if one was given), or the palette stored within the image (if any), or else NULL.
- The **progress_func** call-back occurs after each line of pixels has been read. It is typically used to update some area on the screen showing how much of the image has now been read.
- The **rendering_func** call-back occurs after each line of pixels has been read, just after the progress function. It is typically used to copy the current line of pixels to the screen. The `row` and `row_height` fields will be set before calling this function, and report which horizontal line in the image has just been read.
- The **success_func** call-back is called after all image processing has finished successfully.

If an error occurs part-way through processing, some of the call-backs may not occur at all. For instance, the success function only happens if the image has been completely processed. Any of the call-backs can be set to NULL (which they are initialised to in any case), which prevents that function from being called.

A programmer-specified data pointer can be set in the `user_data` field, for use during the call-back functions. The pointer is never touched by the image reader code.

The remaining fields of the *ImageReader* structure are modified automatically during image processing.

The `state` field begins (and ends) at `STOPPED` and is set to different values as image reading progresses: `STARTING` means that data structures are being allocated, `DITHERING` means the reader is dithering the image to the required palette, `RENDERING` means lines of pixels are being read from the image. The

`IMAGE_ERROR` state only happens if there is an error in the image, or if the connection to the image's file is somehow broken.

The `stage` field reports the current stage in processing, from 1 to `max_stages` inclusive. Different image formats will have a different maximum number of stages. Interlaced images, for example, may have between 3 and 7 stages, depending on the interlacing technique. Dithering may also be counted as a stage. In general, it is not possible to determine in advance how many stages there are for a given image, but it will either be 1, or a small integer usually less than 10.

The `row` gives the current pixel line which is being decoded. This will be a number greater than or equal to zero, and less than the pixel height of the image.

The `row_height` field reports the pixel height of this line of pixels when drawn. Usually this will be equal to one pixel, but it could be larger in the case of interlacing, where lines are read in a non-linear order. For example, a GIF image might store lines in the file in the order 0,4,2,1. In that case, it might be desirable to draw the line taller than 1 pixel, so that the image appears as a series of filled rectangles. Ignoring `row_height` and using a height of 1 for each line in an interlaced image will instead produce a Venetian blind effect.

The `rows_done` field increases from 0 to the image height, inclusive. It is a cumulative total of the number of lines read so far.

The lines of image pixels are stored into either the `data8` or `data32` fields, depending on whether the `required_depth` field was set to 8 or 32.

The `pal` field will point to the image's palette if the `required_depth` was 8, NULL otherwise. This will either be a copy of the `src_pal` supplied by the programmer, or it will be the palette given in the image file, or a constructed palette if the image file contains 24-bit or 32-bit data. The `max_cmap_size` field given by the programmer can be zero, which means the palette can have the maximum possible size (256 elements), or it can be set to a positive integer between 1 and 256 inclusive, to specify the maximum number of colours that may be present in the final image.

## 3.12 Cursors

**OBJECTS**

```
typedef struct Cursor  Cursor;

struct Cursor {
    App *  app;                    /* associated App */
    void * extra;                  /* platform-specific data */
};
```

**FUNCTIONS**

```
Cursor *new_cursor(App *app, Image *img, Point hotspot);
Cursor *get_standard_cursor(App *app, int shape);
void    del_cursor(Cursor *c);
void    find_best_cursor_size(App *app, int *width, int *height,
                              int *depth);
void    set_window_cursor(Window *win, Cursor *c);
Point   get_cursor_position(App *app);
void    set_cursor_position(App *app, Point p);
```

**CONSTANTS**

```
enum StandardCursors {
    BLANK_CURSOR,
    ARROW_CURSOR,
    WAIT_CURSOR,
    CARET_CURSOR,
    CROSS_CURSOR,
    HAND_CURSOR,
    GRAB_CURSOR,
    POINTING_CURSOR,
    PENCIL_CURSOR,
    LASSO_CURSOR,
    DROPPER_CURSOR,
    MAGNIFY_CURSOR,
    MAGPLUS_CURSOR,
    MAGMINUS_CURSOR,
    TEXT_CURSOR      = CARET_CURSOR
};
```

**NOTES**

A *Cursor* represents a mouse pointer on the screen.

New cursors can be created using the **new_cursor** function. The hotspot point is where the 'tip' of the mouse pointer should be, and the image defines the shape of the cursor. The image can have transparent regions, and is allowed to use BLACK, WHITE and CLEAR colours, and can be at least 16 by 16 pixels in size. Other colours and larger cursors sizes may be possible, depending on the window manager.

A number of standard cursors can be obtained using the **get_standard_cursor** function and supplying one of the constants:

- The **BLANK_CURSOR** produces an invisible cursor.
- The **ARROW_CURSOR** is the normal mouse pointer.
- The **WAIT_CURSOR** indicates the user must wait for a short time, for example when loading a large file.
- The **CARET_CURSOR** is used when typing text, and is the same as the **TEXT_CURSOR**.
- The **CROSS_CURSOR** is used when dragging a rectangular region.
- The **HAND_CURSOR** and **GRAB_CURSOR** can be used when dragging a page around.
- The **POINTING_CURSOR** can be used when selecting an item from a list, or clicking on a hyperlink.
- The **PENCIL_CURSOR**, **LASSO_CURSOR** and **DROPPER_CURSOR** can be used during interactive drawing operations, to draw dots or freehand lines, to select a freehand region, or to select a colour from an image, respectively.
- The **MAGNIFY_CURSOR** can be used when increasing or decreasing the magnification of a document. The **MAGPLUS_CURSOR** includes a '+' within the magnifying glass shape, indicating increasing magnification, while the **MAGMINUS_CURSOR** includes a '-' and indicates decreasing magnification.

The memory used by a cursor can be reclaimed by calling **del_cursor**. Normally this is not necessary, since all used cursors are released when the App is deleted.

The **find_best_cursor_size** function asks the window manager what other colours and sizes can be used by a cursor. Pointers to integers are passed, and filled with appropriate values. The width and height will be the maximum size of a cursor. Often these will be 32 or 64, although a cursor might only look good at a width and height of 16, so beware of using all the available space. The depth may be 1, indicating BLACK and WHITE and CLEAR are the only colours to be used, or it may be 32, indicating the cursor may use any colours. The window manager may or may not allow alpha-blending effects to produce shadows around the cursor.

To change a window's cursor, use **set_window_cursor**. This sets the cursor for the entire window. It is valid to call this repeatedly within mouse event handlers.

The **get_cursor_position** returns the current location of the mouse cursor's hotspot in screen co-ordinates. To move the cursor to another location on the screen, use **set_cursor_position** and pass the new position in screen co-ordinates.

# Chapter 4

# Windows and Controls

## 4.1 Windows

### OBJECTS

```
typedef struct Window  Window;

typedef void (*WindowFunc)      (Window *w);
typedef void (*WindowMouseFunc) (Window *w, int buttons, Point xy);
typedef void (*WindowKeyFunc)   (Window *w, unsigned long key);
typedef void (*WindowDrawFunc)  (Window *w, Graphics *g);

struct Window  {
  App *           app;          /* system connection */
  char *          text;         /* title bar string */
  long            flags;        /* status flags */
  Rect            area;         /* drawable area */
  void *          data;         /* user-defined pointer */
  Palette *       pal;          /* private colour palette */
  int             num_children; /* list of child controls */
  Control **      children;
  WindowFunc      close;        /* the user closed the window */
  WindowFunc      resize;       /* the user resized the window */
  WindowDrawFunc  redraw;       /* some part(s) exposed */
  WindowMouseFunc mouse_down;   /* mouse button clicked */
  WindowMouseFunc mouse_up;     /* mouse button released */
  WindowMouseFunc mouse_drag;   /* mouse moved, button down */
  WindowMouseFunc mouse_move;   /* mouse moved, no buttons down */
```

```
  WindowKeyFunc   key_down;     /* Unicode key press */
  WindowKeyFunc   key_action;   /* arrow keys, function keys, etc */
};
```

**FUNCTIONS**

```
Window *new_window(App *app, Rect area, char *name, long flags);
void    del_window(Window *w);

void    show_window(Window *w);
void    hide_window(Window *w);

void    move_window(Window *w, Rect r);
void    size_window(Window *w, Rect r);
void    redraw_rect(Window *w, Rect r);
void    draw_window(Window *w);
void    redraw_window(Window *w);
Rect    get_window_area(Window *w);

void    set_window_title(Window *w, char *title);
char *  get_window_title(Window *w);
void    set_window_icon(Window *win, Image *icon);

void    on_window_close (Window *w, WindowFunc close);
void    on_window_resize(Window *w, WindowFunc resize);
void    on_window_redraw(Window *w, WindowDrawFunc redraw);

void    on_window_mouse_down(Window *w, WindowMouseFunc mouse_down);
void    on_window_mouse_up  (Window *w, WindowMouseFunc mouse_up);
void    on_window_mouse_drag(Window *w, WindowMouseFunc mouse_drag);
void    on_window_mouse_move(Window *w, WindowMouseFunc mouse_move);
void    on_window_key_down  (Window *w, WindowKeyFunc key_down);
void    on_window_key_action(Window *w, WindowKeyFunc key_action);

void    set_window_background(Window *w, Colour bg);
Colour  get_window_background(Window *w);

void    set_window_data(Window *w, void *data);
void *  get_window_data(Window *w);

void    hide_all_windows(App *app);
void    del_all_windows(App *app);
```

**CONSTANTS**

```
#define SIMPLE_WINDOW   0x00000000L

#define MENUBAR         0x00000010L
#define TITLEBAR        0x00000020L
#define CLOSEBOX        0x00000040L
#define RESIZE          0x00000080L
#define MAXIMIZE        0x00000100L
#define MINIMIZE        0x00000200L

#define MODAL           0x00001000L
#define FLOATING        0x00002000L
#define CENTERED        0x00004000L
#define CENTRED         0x00004000L

#define STANDARD_WINDOW (TITLEBAR|CLOSEBOX|RESIZE|MAXIMIZE|MINIMIZE)
```

**NOTES**

A *Window* is a rectangular area displayed on a screen. A window has a zero origin in its own co-ordinate system, but may have various structures built around its drawable area, such as title bars, borders and menu bars. A window can be drawn to by obtaining an appropriate *Graphics* object.

The **new_window** function creates an initially invisible window with the given name. The app parameter specifies the connection to the windowing system. The rectangle specifies where the window's drawable area should appear on the screen, with zero being the top-left point of the screen. The window manager is free to ignore the x and y components of this rectangle, depending on its window placement policies, but most window managers honour these values and try to place the window in the correct place. If the width and height values are too large, the window manager may reduce the size of the window. The flags parameter is usually the constant **STANDARD_WINDOW**. If an error occurs the function returns NULL.

The flags field supplied to **new_window** is a bit-field. Various constants can be combined using the plus or bitwise-or operators to specify how the window should look. Here is a list of those constants and their meanings:

- The **TITLEBAR** flag gives the window a title bar which can be used for moving it around the screen and also for displaying the window's name.

- The **MENUBAR** flag can be used to reserve space for a menu bar on the window, if the platform allows this.
- **CLOSEBOX** gives the user a way of closing the window.
- **MAXIMIZE** gives the user a way of increasing the size of the window to its maximum, and
- **MINIMIZE** allows the window to be shrunk to an icon.
- **RESIZE** gives the user a method of changing the size of the window.
- The **CENTERED** or **CENTRED** flag causes the window to appear at the centre of the screen.
- Adding the **MODAL** flag means the window will be in front of all other application windows when it is displayed, and no mouse or keyboard events will be sent to the other windows until it is hidden.
- **FLOATING** windows will appear in front of all other application windows even when not active.
- A **SIMPLE_WINDOW** is a window with no 'decorations' at all.
- The **STANDARD_WINDOW** constant is defined as having the following flags set: **TITLEBAR, RESIZE, CLOSEBOX, MINIMIZE, MAXIMIZE.** It is provided as a convenience, and is sufficient for most uses of **new_window.**

The window manager might not be able to implement all of the above functionality, and some platforms have different policies regarding placement of menu bars, for instance. The flags should be treated as a request to the windowing system, but that request may be partially or completely ignored. For instance, a window manager might force all windows to have a title bar, whether one was requested or not.

The **del_window** function destroys the specified window, hiding it first if it is currently visible. If a window can be re-used, it is more efficient to hide it and then show it later on, rather than delete the window and recreate it every time the user needs the window.

The **show_window** function makes the specified window visible on the screen and ensures it is the frontmost application window. The **hide_window** function causes the specified window to vanish from the screen. These functions do not destroy

the window, so the window can be shown and hidden many times. This is faster than deleting and recreating a window.

The **set_window_title** function changes the name of the window as shown in the window's title bar, and **get_window_title** returns the current title. Titles must currently be zero-terminated C strings, not UTF-8 encoded Unicode strings, since many window managers are not yet Unicode aware.

The **set_window_icon** function associates an icon image with a window. This icon may be visible when the window is minimised, or in other circumstances. An icon can use colour and transparent pixels, although some window managers, notably under X11, will only display the icon as monochrome. Icons should be at least 32 pixels tall and wide, and might be automatically scaled, cropped or centered to fit the window manager's expectations.

Use **move_window** to change the window's top-left location without changing the size of the window. The supplied rectangle's width and height parameters are ignore by this function.

Use **size_window** to change the window's size without changing the location of the window. The supplied rectangle's x and y parameters are ignore by this function.

The **redraw_rect** function just forces a redrawing of the given rectangle (in window co-ordinates), while the **draw_window** function forces the entire window to be drawn. The **redraw_window** function is the same as **draw_window** except that the existing window contents are first erased using the window's background colour.

The **get_window_area** function returns the window's drawable rectangle in window co-ordinates; hence the top-left point will be zero.

The **on_window_close** function sets the call-back to be used when the user tries to close the window using the window's close-box. If this call-back is not set, the window will simply be hidden. If the call-back is supplied, it will be called instead, and the window will not be hidden. It is then up to the programmer to achieve the desired effect.

The **on_window_resize** function sets the call-back to be used when a window is resized by the user. The window is always *resized* before being *redrawn*, in circumstances where both of these events must be occur.

The **on_window_redraw** function is used to attach a call-back function to a window, which will be called every time that window needs to be redrawn. There is no need for this call-back to clear the window since this will automatically be done by the

window manager.

Mouse events are handled busing the various **on_window_mouse** functions to set call-back functions. A **mouse_down** occurs when a mouse button is clicked within the window; a **mouse_up** occurs when a mouse button is released. Mouse ups may occur outside the window since the event mechanism tracks the mouse even if it leaves the window where a mouse down first occurred. A **mouse_drag** occurs when a mouse button is held down and then the mouse is moved, while a **mouse_move** occurs when no buttons are held down.

Keyboard events are handled using call-back functions. The **on_window_key_down** function sets the handler to be used for most keyboard events, except for the arrow keys, function keys, home, end, page up, page down, insert and delete keys, or keys modified by holding down CONTROL, which are instead handled by **on_window_key_action**. Normal key events are mapped to Unicode values, where possible, and passed as unsigned long integers to the call-back function. Not all operating systems allow the input of Unicode values.

Note that mouse and keyboard events which are intercepted first by a *Control* may not be seen by the underlying window. A control is a separate area placed on a window's surface, which can have its own mouse and keyboard handler functions.

When a window is redrawn, empty areas are first filled with the colour white, by default. A different background colour can be used to fill empty areas, by using the **set_window_background** function. Which colour is currently selected for use as a background can be determined by using **get_window_background**.

A user-supplied pointer may be associated with a window using **set_window_data**. This pointer is converted to a pointer to void and stored in the window's data structure, for later use by the programmer. It can be retrieved using **get_window_data**.

The **hide_all_windows** function hides every window created using the given app parameter, while **del_all_windows** hides and then deletes all such windows. This can be used when terminating the application. The **stop** function automatically calls these when stopping the application anyway, so they may not be needed in many cases.

**EXAMPLES**

- smiley.c
- scribble.c

## 4.2 Labels

### FUNCTIONS

```
Control * new_label(Window *w, Rect r, char *text, int alignment);
Control * add_label(Control *c, Rect r, char *text, int alignment);
```

### CONSTANTS

```
enum {
  ALIGN_LEFT     = 1,
  ALIGN_RIGHT    = 2,
  ALIGN_JUSTIFY  = 3,
  ALIGN_CENTER   = 4,
  ALIGN_CENTRE   = 4,
  VALIGN_TOP     = 8,
  VALIGN_BOTTOM  = 16,
  VALIGN_JUSTIFY = 24,
  VALIGN_CENTER  = 32,
  VALIGN_CENTRE  = 32
};
```

### NOTES

The **new_label** function creates on the specified window a text label which cannot be edited by the user. The text string is aligned within the rectangle according to the value of the alignment parameter. Values such as **ALIGN_LEFT**, **ALIGN_RIGHT** or **ALIGN_CENTER** can be specified. A value of zero corresponds to **ALIGN_LEFT**+**VALIGN_TOP**.

The **add_label** function works in the same way as **new_label**, except that it attaches the label to a control rather than directly to a window.

A label does not respond to mouse or keyboard events; in fact, these events pass straight through a label to the underlying parent control or window. The initial background colour of a label is transparent, so that only the text of the label is drawn against the background of the window.

It can be seen from the list of possible text alignments above, both American and British spellings of the the word 'centre' are legal.

The horizontal alignments are:

- **ALIGN_LEFT** (the default) makes the text start at the left hand edge of the rectangle.
- **ALIGN_RIGHT** causes the text to appear at the rightmost edge of the rectangle. This is useful in fill-in forms.
- **ALIGN_JUSTIFY** is similar to ALIGN_LEFT, except if the text spans more than one line, in which case the spaces between words will be enlarged so each line fills the rectangle horizontally.
- **ALIGN_CENTER** or **ALIGN_CENTRE** have the effect of horizontally centering the text within the rectangle.

The vertical alignments are explained below:

- **VALIGN_TOP** (the default) means the label's text will be drawn starting from the top of the label's rectangle.
- **VALIGN_BOTTOM** draws the text so that it fills the label from the bottom of its rectangle.
- **VALIGN_JUSTIFY** vertically justifies the text, so the inter-line spacing will be large enough for the text to fill the label's rectangle.
- **VALIGN_CENTER** or **VALIGN_CENTRE** vertically centers the text within the label's rectangle.

Either a vertical, or a horizontal alignment can be specified, or a combination of the two, using the plus or bitwise-or operators to combine the alignments.

**EXAMPLES**

- scribble.c

## 4.3 Image Labels

### FUNCTIONS

```
Control *new_image_label(Window *w, Rect r, Image *img, int align);
Control *add_image_label(Control *c, Rect r, Image *img, int align);

void     set_control_image(Control *c, Image *img);
Image *  get_control_image(Control *c);
```

### NOTES

The **new_image_label** function creates a control, which has the specified *Image* drawn within it. The control will appear on the specified window, at the rectangle given in window co-ordinates. The image is not copied by this function, so the image should not be released from memory or modified while it is being used by the control.

The image label control does not respond to user clicks or key events. It merely draws the specified image within its boundaries. The alignment of the image within the rectangle follows the alignment flags described in the entry for labels, except that the image will be scaled to fit the rectangle if an alignment of ALIGN_JUSTIFY or VALIGN_JUSTIFY is given.

The **add_image_label** function works in the same way as **new_image_label**, except that it attaches the image label to a control rather than directly to a window.

To change the image used by a control, call **set_control_image**, and to retrieve a pointer to the currently displayed image, use **get_control_image**.

## 4.4 Buttons

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_button(Window *w, Rect r, char *text, ControlFunc fn);
Control *add_button(Control *c, Rect r, char *text, ControlFunc fn);
```

### NOTES

The **new_button** function creates a push-button control, with the given text string being centered within the button. The button will appear on the specified window in the rectangle, given in window co-ordinates.

When the user clicks on the button with the mouse, the specified call-back function `fn` is called. The parameter to this function will be a pointer to the button which called the function.

The **add_button** function works in the same way as **new_button**, except that it attaches the button to a control rather than directly to a window.

## 4.5 Image Buttons

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_image_button(Window *w, Rect r, Image *i, ControlFunc fn);
Control *add_image_button(Control *c, Rect r, Image *i, ControlFunc fn);

void     set_control_image(Control *c, Image *img);
Image *  get_control_image(Control *c);
```

### NOTES

The **new_image_button** function creates a push-button control, which has the specified *image* drawn within it. The button will appear on the specified window, in the rectangle given in window co-ordinates. The image is not copied by this function, so the image should not be released from memory or modified while it is being used by the control.

When the user clicks on the button with the mouse, the specified call-back function `fn` is called. The parameter to this function will be the button which called the function.

If the button is disabled, an algorithm is used to generate a 'greyed out' image based on the original image.

The **add_image_button** function works in the same way as **new_image_button**, except that it attaches the button to a control rather than directly to a window.

To change the image used by a button, call **set_control_image**, and to retrieve a pointer to the image currently being used by the button, use **get_control_image**.

## 4.6 Check Boxes

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_check_box(Window *w, Rect r, char *text,
                           ControlFunc fn);
Control *add_check_box(Control *c, Rect r, char *text,
                           ControlFunc fn);

int      is_checked(Control *c);  /* is it checked? */
void     check(Control *c);       /* check the check box */
void     uncheck(Control *c);     /* uncheck the check box */
```

### NOTES

The **new_check_box** function creates a *check box* (a square box with text displayed to its right). Clicking on the box causes an `X` to appear within the box, and clicking on it again causes the `X` to vanish.

Each time the state changes between checked and unchecked, the call-back function `fn` is called after the event and the check box is passed to the function as its parameter. This call-back function can be set to NULL, which means no function should be called in response to checking or unchecking the check box.

The **add_check_box** function works in the same way as **new_check_box**, except that it attaches the check box to a control rather than directly to a window.

The function **is_checked** can be used to determine if the check box is currently checked. The function **check** can be used to check a check box, and **uncheck** can be used to remove a check-mark from the check box.

A check box can be freely switched on or off by the user, unlike radio buttons, which are part of mutually exclusive sets. See the section on radio buttons for more details.

### EXAMPLES

- tester.c

## 4.7 Radio Buttons and Radio Groups

**OBJECTS**

```
typedef void (*ControlFunc)  (Control *c);
```

**FUNCTIONS**

```
Control *new_radio_button(Window *w, Rect r, char *text,
                          ControlFunc fn);
Control *add_radio_button(Control *c, Rect r, char *text,
                          ControlFunc fn);

Control *new_radio_group(Window *w);
Control *add_radio_group(Control *c);

int      is_checked(Control *c);  /* is it checked? */
void     check(Control *c);       /* check the radio button */
void     uncheck(Control *c);     /* uncheck the button */
```

**NOTES**

The **new_radio_button** function creates a *radio button* control, which is similar to a check box except that it has a circle which is filled in with a large dot when the user clicks with the mouse.

The **add_radio_button** function works in the same way as **new_radio_button**, except that it attaches the radio button to a control rather than directly to a window.

Whenever the user changes the checked-state of a radio button, the action function `fn` is called after the change. Often this parameter can be left as NULL, which signifies that no function need be called in response to a change of state; instead the state can be determined at some later stage.

One difference between check boxes and radio buttons is that a check box can be freely switched on and off by the user, while radio buttons automatically belong to a mutually exclusive set of radio buttons, known as a 'radio group'. Activating one radio button will therefore switch off the previously active radio button.

Initially, no radio buttons in a radio group will be checked. If a 'default' option needs to be specified, use the **check** function to check one radio button after creating it.

To begin a new radio group, the **new_radio_group** function can be used. After calling this function, all radio buttons subsequently created attached to the same window will belong to a new mutually exclusive set of buttons. Similarly **add_radio_group** starts a new radio group within a control. Radio groups can exist only within one window or control, so each new window has its own initial radio group, as does each separate control.

The function **is_checked** can be used to determine if a radio button is currently checked. The function **check** can be used to activate a radio button while **uncheck** can be used to remove the dot from a radio button.

## 4.8 Image Check Boxes

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_image_check_box(Window *w, Rect r, Image *img,
                             ControlFunc fn);
Control *add_image_check_box(Control *c, Rect r, Image *img,
                             ControlFunc fn);

int      is_checked(Control *c);  /* is it checked? */
void     check(Control *c);       /* check the check box */
void     uncheck(Control *c);     /* uncheck the check box */
```

### NOTES

The **new_image_check_box** function creates a *check box* which displays an image. This is similar to an image button, except that clicking in the button with a mouse will swap its state between unchecked (normal button appearance) and checked (pressed in appearance). If the control is disabled, a greyed version of the image will be displayed instead.

Each time the control changes between a checked and unchecked state, the call-back function `fn` is called after the event and the image check box is passed to the function as its parameter. This call-back function can be set to NULL, which means no function will be called in response to checking or unchecking the image check box.

The **add_image_check_box** function works in the same way as **new_image_check_box**, except that it attaches the image check box to a control rather than directly to a window.

The function **is_checked** can be used to determine if the image check box is currently checked. The function **check** can be used to change the state to checked, and **uncheck** can be used to restore an unchecked state.

To implement mutually exclusive sets of image check boxes (like radio button groups), the call-back function must uncheck the other image check boxes in each set whenever a new image check box becomes checked.

## 4.9 Scroll Bars

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control * new_scroll_bar(Window *w, Rect r, int max,
                         int size_shown, ControlFunc fn);
Control * add_scroll_bar(Control *c, Rect r, int max,
                         int size_shown, ControlFunc fn);
void   change_scroll_bar(Control *c, int pos, int max,
                         int size_shown);
int    get_control_value(Control *c);
```

### NOTES

The **new_scroll_bar** function creates a *scroll bar* which can be used to scroll through text or to change some value within the program. The scroll bar will fill the given rectangle, and will scroll vertically if the rectangle is taller than it is wide, or horizontally otherwise. Manipulating the scrollbar with the mouse changes the scroll position, which has a minimum value of zero (at the left or top of the scroll bar), and a maximum value given by `max`.

The amount of some value which is displayed by the scrollbar can be set with the `size_shown` argument. When the user clicks in the scroll bar to make the scroll 'thumb' position jump, it will jump by this amount. For instance, in a text window which scrolls vertically, if 24 lines out of an 84 line document is visible at any one time, `size_shown` would be set to 24 and max to 84-24, since the top-most line of text on the screen at any time can only range from zero at the start of the document, to 60 at the end. When the user clicks in the scroll bar (other than on the 'thumb') the scroll position will increase or decrease by 24.

Every time the scroll bar position changes, the call-back function `fn` is called. This function is passed the scroll bar control as a parameter.

The **add_scroll_bar** function works in the same way as **new_scroll_bar**, except that it attaches the scroll bar to a control rather than directly to a window.

The **change_scroll_bar** function changes the values used by the scroll bar. It will reset the scroll bar's position `pos`, its maximum value `max` and the `size_shown` value, and re-draw the control if it is currently visible.

The **get_control_value** function returns the scroll bar's current position value.

## 4.10 List Boxes

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_list_box(Window *w, Rect r, char *list[],
                          ControlFunc fn);
Control *add_list_box(Control *c, Rect r, char *list[],
                          ControlFunc fn);

int     get_list_box_item(Control *c);
void    set_list_box_item(Control *c, int index);
void    change_list_box(Control *c, char **list);
void  reset_list_box(Control *c);
```

### NOTES

The **new_list_box** function creates a *list box*, which displays lines of text from a NULL-terminated array of strings in a scrollable area on screen. Only one line of text can be selected at any one time.

Whenever the user clicks on one of the text strings with the mouse, that string becomes *selected* (is drawn highlighted) and the call-back function `fn` is called. The list box control is passed as a parameter, and its value field specifies which string was selected. Any previous selection is deselected first.

The **add_list_box** function works in the same way as **new_list_box**, except that it attaches the list box to a control rather than directly to a window.

The **get_list_box_item** function returns which item is selected or -1 if none are selected. Selected values range from zero to the number of strings minus one. The value will be -1 if no string is currently selected.

Which string is currently selected in the list box can be changed using the **set_list_box_item** function. This will change the hilighting to reflect the new selected item. Passing -1 to this function will remove all hilighting.

The **change_list_box** function sets a new array of strings to use in the list box, and redraws the list. The list of strings is copied, so modification or deletion of the

passed-in list is possible without affecting the list box. The existing selection and scroll bar positions are retained if possible.

The **reset_list_box** function removes the selection highlighting (if any) and positions the scroll bars of the list box so that the first element is visible.

## 4.11 MenuBars, Menus and MenuItems

**OBJECTS**

```
typedef struct MenuBar     MenuBar;
typedef struct Menu        Menu;
typedef struct MenuItem    MenuItem;

typedef void (*MenuAction)(MenuItem *mi);

struct MenuBar {
  Control *   ctrl;          /* associated control */
  Font *      font;          /* for displaying all text */
  int         align;         /* text alignment,direction */
  int         num_menus;     /* list of menus */
  Menu **     menus;
};

struct Menu {
  MenuBar *   parent;        /* enclosing menubar */
  char *      text;          /* name of the menu */
  int         num_items;     /* list of menu items */
  MenuItem ** items;
  int         lasthit;       /* which item was chosen */
  Font *      font;          /* for displaying items */
};

struct MenuItem {
  Menu *      parent;        /* enclosing menu */
  char *      text;          /* menu item name */
  int         shortcut;      /* shortcut key */
  int         state;         /* enabled? checked? */
  Menu *      submenu;       /* pop-up menu */
  MenuAction  action;        /* user-defined action */
  void *      data;          /* user-defined data */
  int         value;         /* user-defined value */
  Font *      font;          /* for displaying this item */
  Colour      fg;            /* for displaying text */
};
```

**FUNCTIONS**

```
MenuBar * new_menu_bar(Window *win);
void      del_menu_bar(MenuBar *mb);

Menu *    new_menu(MenuBar *mb, char *name);
Menu *    new_sub_menu(Menu *parent, char *name);
void      del_menu(Menu *m);

MenuItem *new_menu_item(Menu *m, char *name, int key,
                            MenuAction fn);
void      del_menu_item(MenuItem *mi);

void      check_menu_item(MenuItem *mi);
void      uncheck_menu_item(MenuItem *mi);
int       menu_item_is_checked(MenuItem *mi);

void      enable_menu_item(MenuItem *mi);
void      disable_menu_item(MenuItem *mi);
int       menu_item_is_enabled(MenuItem *mi);

void      set_menu_item_value(MenuItem *mi, int value);
int       get_menu_item_value(MenuItem *mi);

void      set_menu_item_foreground(MenuItem *mi, Colour col);
Colour    get_menu_item_foreground(MenuItem *mi);

void      set_menu_item_font(MenuItem *mi, Font *font);
Font *    get_menu_item_font(MenuItem *mi);

void      set_menu_foreground(Menu *m, Colour col);
Colour    get_menu_foreground(Menu *m);

void      set_menu_font(Menu *m, Font *font);
Font *    get_menu_font(Menu *m);

void      set_menu_bar_font(MenuBar *mb, Font *font);
Font *    get_menu_bar_font(MenuBar *mb);
```

**NOTES**

A *MenuBar* is a horizontal bar in which the names of menus appear. A *Menu* refers to a pull-down menu which contains *MenuItems*.

A menu bar can be associated with a window by calling **new_menu_bar** after

that window has been created. This function will return NULL if it fails for some reason. Calling **del_menu_bar** will destroy the menu bar and all associated menus and menu items, also removing the menu bar from the window. Destroying a window will automatically call this function.

After creating a menubar, the **new_menu** function is used to create menus, which are attached to the menubar. Each menu has a `name` which is displayed in the menubar, for example "File" might be the name of the menu which controls file operations.

The **new_sub_menu** function can also be used to create menus. The `parent` parameter specifies a menu to which the new sub-menu will be added. The `name` of the sub-menu will appear in the `parent` menu, and selecting this name will make the sub-menu appear. The **del_menu** function can be used to delete a menu from a menu bar, and destroy all of its menu items.

After the creation of a menu, menu items can be added to it using **new_menu_item**. The `name` of the item is a string which is displayed in the menu. Next to that name a short-cut key can be displayed. If `key` is non-zero it specifies an ASCII character which can be typed in combination with some other keyboard key to activate this menu item. For instance, if the key were given as 'Q', on a Windows platform the menu item would appear with "Ctrl+Q" next to its name, signifying that pressing the control key and the letter 'Q' together would trigger this menu item. Other platforms would display their own normal menu key-combinations.

A menuitem created with a name which is a hyphen "-" will cause a 'separator' line to appear in the pull-down menu. This can be used to logically group items in a menu.

The function pointer `fn` given as a parameter to **new_menu_item** is called when that menu item is selected by the user. When this call-back function is called, a pointer to the menu item is passed as a parameter to `fn`.

Use **del_menu_item** to delete one menu item and remove it from its menu.

The **check_menu_item** function places a check-mark beside a menu item within its menu, while the **uncheck_menu_item** function removes any such check-mark. The **menu_item_is_checked** function returns non-zero if the menu item has a check-mark next to it, zero if it does not. Menu items initially have no check-mark.

Menu items are enabled for mouse selection by default. They can be disabled using **disabled_menu_item**, and re-enabled using **enable_menu_item**. Whether a menu item is currently enabled can be determined using **menu_item_is_enabled**.

Separator menu items are disabled when created.

Each menu item can have an integer value associated with it, by **set_menu_item_value**. The value can be retrieved using **get_menu_item_value**.

A menu item can also have its own font and foreground colour, which are used when the font's text is to be displayed. The functions **set_menu_item_foreground** and **get_menu_item_foreground** access the text colour.

The functions **set_menu_item_font** and **get_menu_item_font** are used to access a menu item's text font. By default, a menu item does not have its own font, in which case the menu bar's font is used for displaying text, which is Unifont by default. Setting the menu bar's font using **set_menu_bar_font** is all that is needed to change all menu items' displayed text to a single font.

Menus can also have a text font, accessed with **set_menu_font** and **get_menu_font**. A menu's font is only used when displaying its name in a menu bar or as a sub-menu name within another menu. If it has no font, the menu bar's font is used. Similarly, its text colour is accessed with **set_menu_foreground** and **get_menu_foreground**, and is only used for displaying the menu's name.

## 4.12 Drop-down Lists

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_drop_list(Window *w, Rect r, char **lines,
                           ControlFunc fn);
Control *add_drop_list(Control *c, Rect r, char **lines,
                           ControlFunc fn);

long       get_control_value(Control *c);
```

### NOTES

A *drop list* is a push-button which, when clicked, displays a drop-down menu list of selectable names. The control is created by **new_drop_list** on the specified window, within the given rectangle. The `lines` parameter is a NULL-terminated list of strings, to be displayed in the menu. The function pointer `fn` will be called when the user chooses one of the lines from the menu.

The **add_drop_list** function works in the same way as **new_drop_list**, except that it attaches the list to a control rather than directly to a window.

The program can determine which item was chosen by examining the control's value, using **get_control_value**. The first string in the list is numbered 0, the next 1, and so on.

### EXAMPLES

- scribble.c

## 4.13 Drop-Fields

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_drop_field(Window *w, Rect r, char **lines);
Control *add_drop_field(Control *c, Rect r, char **lines);

char *   get_control_text(Control *c);
```

### NOTES

A *drop field* is a one-line text field with an associated push-button which, when clicked, displays a drop-down menu list of selectable names. The control is created using **new_drop_field** on the specified window, within the given rectangle. The `lines` parameter is a NULL-terminated list of strings, to be displayed in the menu.

Text can be typed as normal into the text field, or an item can be selected from the drop-down menu, which will then set the text field's text to be the selected item.

The **add_drop_field** function works in the same way as **new_drop_field**, except that it attaches the drop field to a control rather than directly to a window.

The **get_control_text** function can be used to discover what text was typed or selected.

### EXAMPLES

- alldemo.c

## 4.14 Pop-up Lists

### FUNCTIONS

```
int  pop_up_list(Window *w, Font *f, char **lines,
                     int buttons, Point p);
```

### NOTES

A *pop-up list* is a menu which appears at the press of a mouse button and contains a list of selectable names. The menu will appear on the specified window, and vanish when the user releases the mouse button which was used to invoke the menu. The given font is used for menu text, unless it is NULL, in which case the default font is used.

The `lines` parameter is a NULL-terminated list of strings, to be displayed in the menu. The `buttons` parameter specifies which mouse button(s) were used to invoke the menu. Constants such as LEFT_BUTTON or RIGHT_BUTTON can be used here. The menu will appear near the given point.

The function returns -1 if no list item was chosen when the mouse button was released, or else a number between 0 and the number of strings in the list minus one, indicating which item was chosen.

### EXAMPLES

- imagine.c

## 4.15 Text Fields

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_field(Window *w, Rect r, char *text);
Control *add_field(Control *c, Rect r, char *text);

void     set_control_text(Control *c, char *newtext);
char *   get_control_text(Control *c);

void     set_focus(Control *c);

void     set_field_allowed_width(Control *c, int width);
void     set_field_allowed_chars(Control *c, char *allowed);
void     set_field_disallowed_chars(Control *c, char *disallowed);
```

### NOTES

The **new_field** function creates an editable text *field*, with the specified text string displayed within it. The text field will be displayed on one line only and will scroll horizontally if the user enters more text into it than will fit within the rectangle.

The **add_field** function works in the same way as **new_field**, except that it attaches the field to a control rather than directly to a window.

The text inside a field can be changed using the **set_control_text** function. This function makes a copy of the string and stores it into the text field.

Use **get_control_text** to find the current text inside a textbox or field. The string returned from **get_control_text** is a read-only string, so care must be taken to avoid modifying or deleting it.

Note that keyboard events will not be sent to a field unless it currently has keyboard focus. The **set_focus** function can be used to set a window's key focus to a given control. Setting focus will remove the focus from whichever control previously had it, if any.

The maximum number of characters which can be typed into a field can be controlled with the **set_field_allowed_width** function. The `width` parameter controls

the number of Unicode characters which can be typed. If the value is zero, the field is unrestricted in width, which is the default.

A field can be restricted with **set_field_allowed_chars**, so that only a set of allowed characters can be typed into the field by the user. All other characters will be passed to the parent control or window. Setting the `allowed` characters to the UTF-8 encoded string "0123456789", for example, restricts a field so it only accepts digits, while setting the allowed string to "0123456789.$" additionally allows the decimal point and dollar sign. Setting the allowed string to NULL removes all restrictions, which is the default situation.

A field can also be instructed to ignore certain characters, and instead pass them up to its parent control or window, using the **set_field_disallowed_chars** function. For example, if the `disallowed` string was set to "\n\t", the enter and tab keys could not be typed into a field, and could thus propagate to the window, where a key handler function could decide to shift focus, or perform some other action as a result. By default, the tab, newline and escape keys are disallowed for fields, but the tab and newline keys are allowed in text boxes, so this function is particularly useful for controlling multi-line text boxes.

## 4.16 Password Fields

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_password_field(Window *w, Rect r, char *text);
Control *add_password_field(Control *c, Rect r, char *text);

void     set_control_text(Control *c, char *newtext);
char *   get_control_text(Control *c);

void     set_focus(Control *c);
```

### NOTES

The **new_password_field** function creates an editable password text *field*, with the specified text string displayed in a secretive fashion to prevent observers from discovering the password contained within. The text field will be displayed on one line only and will scroll horizontally if the user enters more text into it than will fit within the rectangle.

The **add_password_field** function works in the same way as **new_password_field**, except that it attaches the field to a control rather than directly to a window.

The text inside a field can be changed using the **set_control_text** function. This function makes a copy of the string and stores it into the text field.

Use **get_control_text** to find the actual text inside the field. The string returned from **get_control_text** is a read-only string, so care must be taken to avoid modifying or deleting it.

The cut and copy text operations will not reveal the password text. Instead the clipboard will only contain what is visible in the password field. This prevents a user from copying the password text to clear text elsewhere. The paste operation works normally and can be used to paste a password into the field.

Note that keyboard events will not be sent to a field unless it currently has keyboard focus. The **set_focus** function can be used to set a window's key focus to a given control. Setting focus will remove the focus from whichever control previously had it, if any.

## 4.17 Text Boxes

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_text_box(Window *w, Rect r, char *text);
Control *add_text_box(Control *c, Rect r, char *text);

void     set_control_text(Control *c, char *newtext);
char *   get_control_text(Control *c);

void     set_focus(Control *c);
```

### NOTES

To create a text field which can contain multiple lines of text use the **new_text_box** function. A *text box* has a vertical scrollbar which allows the user to move the text up and down. If a word will not fit on one line of text, that word will 'wrap' onto the next line of text.

The text inside a text box can be changed using the **set_control_text** function. This function makes a copy of the string and stores it into the text box.

The **add_text_box** function works in the same way as **new_text_box**, except that it attaches the text box to a control rather than directly to a window.

Use **get_control_text** to find the current text inside a text box. The string returned is a read-only string, so care must be taken not to free or modify the string.

Text can be cut to the clipboard using Ctrl-X, copied using Ctrl-C, and pasted using Ctrl-V. Arrow keys, mouse and shift-arrow selection, and the delete, home, end, page up and page down keys will all work as expected.

Note that keyboard events will not be sent to a text box unless it currently has keyboard focus. The **set_focus** function can be used to set a window's key focus to a given control. Setting focus will remove the focus from whichever control previously had it, if any.

## 4.18 Text Editing Functions

### FUNCTIONS

```
void  cut_text(Control *c);   /* cut selection to clipboard */
void  copy_text(Control *c);  /* copy selection to clipboard */
void  clear_text(Control *c); /* clear selection */
void  paste_text(Control *c); /* paste over selection */

void  insert_text(Control *c, char *text);
void  select_text(Control *c, long start, long end);
void  text_selection(Control *c, long *start, long *end);

void  set_text(Control *c, char *newtext); /* find contents */
char *get_text(Control *c);                /* change contents */
```

### NOTES

The following text editing functions will work with any *textbox* or *field*:

The **cut_text** function cuts whatever text is currently selected from the specified textbox, and places the text into the clipboard, while **copy_text** copies the text to the clipboard without deleting the text. To delete the currently selected text without transferring it to the clipboard, **clear_text** can be used.

The **paste_text** function pastes the text in the clipboard into the specified textbox, deleting any currently selected text in the process.

The **insert_text** function inserts a specified text string after the insertion point in a textbox. The insertion point is then moved to be after the newly inserted text. Repeated use of this function will thus behave the same as if the user had typed text into the textbox.

The current text selection can be changed by use of **select_text**. The start and end locations are measured from zero being before the first character in the textbox. The number negative one has special meaning: it refers to the location after the last character in the textbox. Hence **select_text(-1,-1)** moves the insertion point to the end of the textbox, **select_text(0,0)** moves the insertion point to the start, and **select_text(0,-1)** selects all the text.

What text is currently selected can be found by use of the **text_selection** function. It returns the start and end-points of any selected text in the supplied long integer pointers.

Many controls have associated with them a text string, (e.g. *buttons, text fields, labels, windows*). This text string can be changed using **set_text** and can be found using **get_text**. The string returned from **get_text** is a read-only string! Do not modify it or free it.

## 4.19 Tab Buttons

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
Control *new_tab_button(Window *w, Rect r, char *text, ControlFunc fn);
Control *add_tab_button(Control *c, Rect r, char *text, ControlFunc fn);
```

### NOTES

The **new_tab_button** function creates a tab-pane button control, with the given text string being centered within the button. The button will appear on the specified window in the rectangle, given in window co-ordinates.

When the user clicks on the tab button with the mouse, the specified call-back function `fn` is called. The parameter to this function will be a pointer to the button which called the function. Usually, this function will bring a control 'pane' to the front within a window, to display that pane, as well as bringing the tab button itself to the front.

The **add_tab_button** function works in the same way as **new_tab_button**, except that it attaches the tab button to a control rather than directly to a window.

# Chapter 5

# Using Controls

## 5.1 Controls

**OBJECTS**

```
struct Control {
  Rect            area;           /* rectangle on parent */
  Point           offset;         /* to window's co-ordinates */
  Window *        win;            /* parent window, if any */
  Control *       parent;         /* parent control, if any */
  int             num_children;   /* list of child controls */
  Control **      children;
  Colour          bg;             /* background fill colour */
  Colour          fg;             /* foreground drawing colour */
  int             state;          /* VISIBLE, CHECKED, etc */
  Region *        visible;        /* in window co-ords */
  void *          data;           /* user-defined pointer */
  char *          text;           /* text to display */
  void *          extra;          /* internal pointer */
  long            value;          /* internal integer value */
  Image *         img;            /* internal image pointer */
  Font *          font;           /* internal image pointer */
  ControlFunc     resize;         /* event handlers */
  DrawFunc        redraw;
  MouseFunc       mouse_down;
  MouseFunc       mouse_up;
  MouseFunc       mouse_drag;
  MouseFunc       mouse_move;
```

```
  KeyFunc         key_down;
  KeyFunc         key_action;
  ControlFunc     action;
  ControlFunc     update;
  ControlFunc     refocus;
  ControlFunc     del;
};

typedef void (*ControlFunc)  (Control *c);
```

**FUNCTIONS**

```
Control *new_control(Window *parent, Rect r);
Control *add_control(Ctronol *parent, Rect r);
void     del_control(Control *c);

int      remove_control(Control *c);
int      attach_to_window(Window *parent, Control *c);
int      attach_to_control(Control *parent, Control *c);
int      bring_control_to_front(Control *c);
int      send_control_to_back(Control *c);

Window * parent_window(Control *c);
```

**NOTES**

A *Control* is a area on a window which responds to user's actions, such as mouse clicks and the keyboard, to perform some function. Functions which create controls take as an argument a rectangle, which specifies where on the current window the control should be placed. The rectangle is measured in pixels, relative to the top-left point of the window which is the point (0,0).

Buttons, check boxes, text fields, etc are all kinds of *Controls*, so many of the functions described in the following sections apply equally to those objects, as they do to custom-made controls.

The **new_control** function creates and returns a generic control of the requested size and attaches it to the given window. The control can have call-backs added to it using **on_control_redraw, ap_on_control_mouse_down** etc (see later sections), to allow the control to respond to events.

The **add_control** function behaves much the same, except the created control is attached to the given parent control. Controls can have other controls attached

to their surfaces. These child controls will appear inside the parent control, and cannot move outside of the parent's boundary.

The **del_control** function destroys a control and removes it from any window to which it is attached. Normally this function is not used, since destroying a window will destroy all of its child controls.

The **remove_control** function removes a control from its parent window or control. A side-effect of this function is that the control will become invisible since the parent window or control will be redrawn. Such a removed control can be re-attached to a window or control's surface using **attach_to_window** or **attach_to_control**.

A control's position in the stacking hierarchy can be manipulated. The **bring_control_to_front** function moves a control in front of its siblings, while **send_control_to_back** places it behind its siblings.

The **parent_window** function returns the window where the control resides. If a control is a child of another control, this function looks up the parent window of the parent controls until it finds the enclosing window. If the control or its parents are not currently attached to any window, this function returns NULL.

## 5.2 Changing the Appearance of Controls

**FUNCTIONS**

```
void   set_control_background(Control *c, Colour col);
Colour get_control_background(Control *c);

void   set_control_foreground(Control *c, Colour col);
Colour get_control_foreground(Control *c);

void    set_control_image(Control *c, Image *img);
Image * get_control_image(Control *c);

void    set_control_font(Control *c, Font *f);
Font *  get_control_font(Control *c);
```

**NOTES**

When a control or window is drawn on the screen, its rectangle is first automatically cleared by filling it with its background colour (unless it is set to be the special value **CLEAR**).

The **set_control_background** function takes a *Colour* value as a parameter and sets the background colour of the control to that value. The background colour is returned by **get_control_background**.

After the background is drawn, the foreground colour is used in some places to draw text and other features of a control. The **set_control_foreground** and **get_control_foreground** functions access this foreground colour. It is not guaranteed that the foreground colour will be used by all controls (for example, image labels draw images and don't need to use a foreground colour, while buttons do use the foreground colour when drawing text).

For controls which display an image, the **set_control_image** can be used to change the image displayed, and **get_control_image** returns the current image used.

For controls which display text, the **set_control_font** can be used to change the font used, and **get_control_font** returns the current font used. **EXAMPLES**

- tester.c

## 5.3 Adding Data to Controls

### OBJECTS

```
typedef void (*ControlFunc)  (Control *c);
```

### FUNCTIONS

```
void  set_control_value(Control *c, long value);
long  get_control_value(Control *c);

void  set_control_data(Control *c, void *data);
void *get_control_data(Control *c);

void  on_control_action(Control *c, ControlFunc action);

void  activate_control(Control *c);
```

### NOTES

Every control can have a long integer value associated with it. Some controls, such as scrollbars and listboxes use this integer value to represent the current state of the control.

For programmer-defined controls, the **set_control_value** and **get_control_value** functions exist to allow setting this value, or finding the value.

If a control's state is more complex than an integer, more data must be stored with a control. The **set_control_data** function can store a pointer with a control, which could point to a data structure in memory. To find the value of this data pointer, use **get_control_data**. Note, this function returns the pointer as a void pointer so it is necessary to store this to the appropriate pointer type before use.

A control can have a call-back function associated with it for normal uses. This call-back is usually called by controls (such as buttons) whenever the user clicks on the control.

The function to be called can be set using the **on_control_action** function. This can be used to set or change a button's response to events.

The **activate_control** function will call a control's call-back (if the function pointer is not NULL, which it is by default).

## 5.4 Drawing Controls

**OBJECTS**

```
typedef void (*DrawFunc)(Control *c, Graphics *g);
```

**FUNCTIONS**

```
void  on_control_redraw(Control *c, DrawFunc redraw);

void  draw_control(Control *c);
void  redraw_control(Control *c);
Rect  get_control_area(Control *c);
```

**NOTES**

The **on_control_redraw** function is used to attach a call-back function to a control which will be called every time that control needs to be redrawn. There is no need for this call-back to clear the control's background since this will done automatically. The call-back function should use drawing operations (see later sections) to draw the entire contents of the control.

Built-in controls such as buttons, scrollbars, etc have their own redrawing functions, so it is unwise to use the functions in this section on anything except custom-defined controls.

The **draw_control** function calls the control's redraw call-back, as set using **on_control_redraw** (see above). The redraw call-back, which is supplied by the programmer, is passed two parameters: the control which needs to be redrawn, and a *Graphics* object which can be used to draw to the control's surface (see later sections).

The **redraw_control** function behaves in a similar way, except it first clears the control's visible area using the control's background colour, unless that colour is transparent.

The **get_control_area** function can be used within the drawing call-back, to obtain the rectangle of the control, in its own co-ordinate system. Hence, the top-left point of this rectangle will be (0,0).

## 5.5 Resizing and Moving Controls

**FUNCTIONS**

```
Rect  get_control_area(Control *c);

void  move_control(Control *c, Rect r);
void  size_control(Control *c, Rect r);
```

**NOTES**

The **get_control_area** function can be used to discover the rectangle of a control in its own co-ordinate system. Hence, the top-left point of this rectangle will be (0,0).

A control can be moved using the **move_control** function. This will change the location (but not the size) of the control on its parent. The width and height fields in the supplied rectangle will be ignored, while the x and y field must be expressed relative the parent's top-left point.

A control's size can be changed without moving it, using **size_control**. Here, the given rectangle's width and height field are used, while the x and y fields are ignored. The control's top-left point will remain in the same location.

## 5.6 Hiding Controls

### FUNCTIONS

```
void  show_control(Control *c);
void  hide_control(Control *c);
int   is_visible(Control *c);
```

### NOTES

The **show_control** and **hide_control** functions make a control visible or invisible on its parent. The **is_visible** function returns zero if the control's state indicates it is not visible, or one if it is visible.

By default, controls are visible when they are created, but windows are not. Visibility is the quality of whether the control should be drawn if it is not obscured, *not* whether the control is currently unobscured.

A control might be visible, even if its parent window is not. Every object has its own visibility state, which is separate to every other object's state.

## 5.7 Disabling Controls

### FUNCTIONS

```
void  enable(Control *c);
void  disable(Control *c);
int   is_enabled(Control *c);
```

### NOTES

A control can be enabled so it can receive user input and mouse clicks, or disabled so that it cannot.

To enable a control to receive user input and mouse clicks, call the **enable** function. By default, when a control is created, it is always enabled. However, this function can re-enable a control which has been disabled.

To disable a control, and make it ignore user input, use **disable**. A disabled control will usually have grey text, and will not respond to the user. An exception to this is a disabled textbox or field, which will appear normal, but cannot be edited or modified by the user.

The **is_enabled** function will return zero if the control is disabled, and one if it is enabled.

## 5.8 Hilighting Controls

### FUNCTIONS

```
void  highlight(Control *c);
void  unhighlight(Control *c);
int   is_highlighted(Control *c);

void  flash_control(Control *c);
```

### NOTES

Push buttons, check boxes, radio buttons, etc are highlighted when the user holds down a mouse button inside the control's boundary, and become unhighlighted when the mouse button is released. The **highlight** function makes a control appear highlighted, and **unhighlight** returns a control to its normal appearance. Whether or not a control is currently highlighted is reported by **is_highlighted**.

The function **flash_control** simulates a mouse button click in a button, check box or radio button. The control will become highlighted and after a short period of time will be unhighlighted. This function does nothing more than change the appearance of the control for a brief period of time.

## 5.9 Keyboard Focus

### FUNCTIONS

```
void  set_focus(Control *c);
int   has_focus(Control *c);

void  pass_event(Control *c);
```

### NOTES

A control can have the keyboard focus, which means that in its parent window, if a user types a key on the keyboard, that key click will be sent to the control which has focus rather than to the window itself.

To set focus to a particular control, use **set_focus**. This will remove focus from whatever control previously had the focus, if any, and it will redraw all affected controls (since some controls appear differently if they have the focus).

Use **has_focus** to determine if the given control has the keyboard focus.

Focus has no effect on mouse events.

The **pass_event** function can be called within a call-back function to pass a keyboard event up the object hierarchy (to the enclosing parent control or window). This might be used if a control handles some events, but not others. For instance, a text field handles ordinary key strokes, but might not want to accept 'Enter' or 'Tab' key events; instead it might want to pass these events to the window's call-back functions.

## 5.10 Responding to Mouse Events

**OBJECTS**

```
typedef void (*MouseFunc)(Control *c, int buttons, Point xy);
```

**FUNCTIONS**

```
void  on_control_mouse_down(Control *c, MouseFunc mouse_down);
void  on_control_mouse_up  (Control *c, MouseFunc mouse_up);
void  on_control_mouse_drag(Control *c, MouseFunc mouse_drag);
void  on_control_mouse_move(Control *c, MouseFunc mouse_move);
```

**CONSTANTS**

```
enum {
  NO_BUTTON     = 0,
  LEFT_BUTTON   = 1,
  MIDDLE_BUTTON = 2,
  RIGHT_BUTTON  = 4
};
```

**NOTES**

The functions **on_control_mouse_down**, **on_control_mouse_up**, **on_control_mouse_drag**, **on_control_mouse_move** allow controls to respond to mouse events. They associate call-back functions with a control, so that when a certain kind of mouse events occurs, the relevant call-back function is called.

Each of the functions handles a different kind of mouse event:

- A **mouse_down** occurs when a mouse button is clicked while the mouse pointer is inside a control.
- A **mouse_move** occurs when the mouse is moved within a control, but no mouse buttons are held down at the time.
- A **mouse_drag** occurs when the mouse is moved while one or more mouse buttons are held down at the same time.

- A **mouse_up** occurs when any mouse button is released (some mouse buttons may still be held down).

Each of the call-backs are defined as *MouseFunc*. A MouseFunc has an integer parameter `buttons` and a point `xy` expressed in the control's own co-ordinate system. These refer to the mouse state at the time the event happened.

If any of the mouse buttons are held down when the call-back is activated, the `buttons` parameter will be set to reflect the fact. It is a bit-field which is organised so that `buttons & LEFT_BUTTON` is set when the left mouse button is down, `buttons & MIDDLE_BUTTON` corresponds to the middle mouse button and `buttons & RIGHT_BUTTON` corresponds to the right mouse button. It will be zero if no buttons are held down.

For systems which have a mouse with fewer than three buttons, extra buttons can be simulated by holding down modifier keys and clicking with the left button. Holding down Shift simulates a right mouse button click, while holding down Ctrl simulates a middle button click. Holding down Alt simulates a left mouse button click, which is useful for situations where the user wants several buttons clicked at the same time on a two or one button mouse.

## 5.11 Responding to Keyboard Events

**OBJECTS**

```
typedef void (*KeyFunc)(Control *c, unsigned long key);
```

**FUNCTIONS**

```
void  on_control_key_down  (Control *c, KeyFunc key_down);
void  on_control_key_action(Control *c, KeyFunc key_action);
```

**CONSTANTS**

```
enum {
  BELL  = 0x07,           /* ASCII bell character */
  BKSP  = 0x08,           /* ASCII backspace */
  VTAB  = 0x0B,           /* ASCII vertical tab */
  FF    = 0x0C,           /* ASCII form-feed */
  ESC   = 0x1B            /* ASCII escape character */
};

enum {
  INS   = 0x2041,         /* Insert key */
  DEL   = 0x2326,         /* Delete key */
  HOME  = 0x21B8,         /* Home key */
  END   = 0x2198,         /* End key */
  PGUP  = 0x21DE,         /* Page Up key */
  PGDN  = 0x21DF,         /* Page Down key */
  ENTER = 0x2324          /* Numerical keypad Enter key */
};

enum {
  LEFT  = 0x2190,         /* Left arrow key */
  UP    = 0x2191,         /* Up arrow key */
  RIGHT = 0x2192,         /* Right arrow key */
  DOWN  = 0x2193          /* Down arrow key */
};

enum {
  F1    = 0x276C,         /* Function keys */
```

```
  F2    = 0x276D,
  F3    = 0x276E,
  F4    = 0x276F,
  F5    = 0x2770,
  F6    = 0x2771,
  F7    = 0x2772,
  F8    = 0x2773,
  F9    = 0x2774,
  F10   = 0x2775
};

enum {
  CONTROL = 0x20000000L,  /* Modifier bit-fields */
  SHIFT   = 0x10000000L
};
```

**NOTES**

Keyboard events can be caught using the **on_control_key_down** and **on_control_key_action** functions. The first function sets the call-back used to handle normal Unicode keys, including the return and escape keys. Keys which do not produce Unicode values are sent to the second function's call-back, such as when the user presses an arrow key or a function key.

Keyboard call-backs can be associated with controls. Typically you should not use these functions with pre-defined controls such as buttons, fields etc, only with custom-designed controls, since the pre-defined controls already make use of their own keyboard call-backs.

The call-back specified by the **on_control_key_down** function may be passed any of the following keys:

- Any normal ASCII key value which can generated by the keyboard.
- The value **ESC** is defined as the Escape key.
- The value **BKSP** is defined as the Backspace key.
- A newline character `'\n'` is sent when the main keyboard Enter or Return key is pressed (but not the numeric keypad Enter key).
- Any Unicode value which the keyboard may generate.

To see other keyboard events, the **on_control_key_action** function associates a call-back with a control or window. That call-back can see the following key values:

- **INS, DEL, HOME, END, PGUP, PGDN** which correspond to the keys Insert, Delete, Home, End, Page Up and Page Down.
- **ENTER** which is the Enter key on a numeric keypad. Note that this value is different from the normal ASCII return key, `0x0A` or `'\n'` (newline).
- **LEFT, UP, RIGHT, DOWN** which are the arrow keys.
- **F1, F2, F3, F4, F5, F6, F7, F8, F9, F10** which correspond to the Function keys on a keyboard. Note that there is currently no provision for Function keys higher than **F10**.
- Keys modified by holding down the **CONTROL** key, or the other keys in this list modified by **CONTROL** and/or **SHIFT** being held down.

These keys should not be handled by same call-back as ordinary **key_down** events, since a Unicode keyboard may be able to generate the same value explicitly, and then confusion would exist about whether the user pressed an arrow key, or generated a Unicode value which happens to map to the same number.

## 5.12 Summary of Control Functions

**FUNCTIONS**

```
Control *new_control(Window *parent, Rect r);
Control *add_control(Control *parent, Rect r);
void     del_control(Control *c);

int     attach_to_window(Window *parent, Control *c);
int     attach_to_control(Control *parent, Control *c);
int     remove_control(Control *c);
int     bring_control_to_front(Control *c);
int     send_control_to_back(Control *c);

Window *parent_window(Control *c);

void    draw_control(Control *c);
void    redraw_control(Control *c);

Rect    get_control_area(Control *c);
void    move_control(Control *c, Rect r);
void    size_control(Control *c, Rect r);

int     is_visible(Control *c);
void    show_control(Control *c);
void    hide_control(Control *c);

void    set_control_background(Control *c, Colour col);
Colour  get_control_background(Control *c);

void    set_control_foreground(Control *c, Colour col);
Colour  get_control_foreground(Control *c);

void    set_control_text(Control *c, char *text);
char *  get_control_text(Control *c);

void    set_control_data(Control *c, void *data);
void *  get_control_data(Control *c);

void    set_control_value(Control *c, long value);
long    get_control_value(Control *c);

void    set_control_image(Control *c, Image *img);
```

```
Image * get_control_image(Control *c);

void    set_control_font(Control *c, Font *f);
Font *  get_control_font(Control *c);

int   is_enabled(Control *c);
void  enable(Control *c);
void  disable(Control *c);

int   is_checked(Control *c);
void  check(Control *c);
void  uncheck(Control *c);

int   is_highlighted(Control *c);
void  highlight(Control *c);
void  unhighlight(Control *c);

int   is_armed(Control *c);
void  arm(Control *c);
void  disarm(Control *c);

int   has_focus(Control *c);
void  set_focus(Control *c);

void  flash_control(Control *c);
void  activate_control(Control *c);
```

**NOTES**

The above is a list of the functions which will work on many different kinds of controls and windows. See the individual sections for details on each function.

Most of the above functions cause a control's update call-back to be called. If no update call-back has been set, the control is redrawn by default.

When a control is deleted, its deletion call-back is called just prior to deletion, if one exists. This function typically tidies up and released any memory stored in the control's extra pointer.

## 5.13 Event Handlers

### OBJECTS

```
typedef void (*ControlFunc)(Control *c);
typedef void (*MouseFunc)  (Control *c, int buttons, Point xy);
typedef void (*KeyFunc)    (Control *c, unsigned long key);
typedef void (*DrawFunc)   (Control *c, Graphics *g);
```

### FUNCTIONS

```
void  on_control_resize    (Control *c, ControlFunc resize)
void  on_control_redraw    (Control *c, DrawFunc redraw);
void  on_control_mouse_down(Control *c, MouseFunc mouse_down);
void  on_control_mouse_up  (Control *c, MouseFunc mouse_up);
void  on_control_mouse_drag(Control *c, MouseFunc mouse_drag);
void  on_control_mouse_move(Control *c, MouseFunc mouse_move);
void  on_control_key_down  (Control *c, KeyFunc key_down);
void  on_control_key_action(Control *c, KeyFunc key_action);
void  on_control_action    (Control *c, ControlFunc action);
void  on_control_update    (Control *c, ControlFunc update);
void  on_control_refocus   (Control *c, ControlFunc refocus);
void  on_control_deletion  (Control *c, ControlFunc del);

void  pass_event(Control *c);
```

### NOTES

Above is a list of the event handling call-back functions which can be set for programmer-defined controls. Some of these functions will do nothing for pre-defined controls such as buttons, check boxes etc. See the individual sections for more details.

The **pass_event** function can be called within a call-back function to pass the event up the object hierarchy. This might be used if a control handles some events, but not others. For instance, a text field handles ordinary key strokes, but might not want to accept 'Enter' or 'Tab' key events; instead it might want to pass these events to the window's call-back functions.

# Chapter 6

# Drawing Operations

## 6.1 Graphics Objects

**OBJECTS**

```
typedef void (*WindowDrawFunc)(Window *w, Graphics *g);
typedef void (*DrawFunc)(Control *c, Graphics *g);

typedef struct Graphics  Graphics;

struct Graphics {
  Colour    colour;     /* current drawing colour */
  Font *    font;       /* current text drawing font */
  int       line_width; /* line width in pixels */
  Window *  win;        /* target window, or */
  Bitmap *  bmap;       /* target bitmap, or */
  Control * ctrl;       /* target control, or */
  Image *   img;        /* target image */
  Region *  clip;       /* clip all drawing to this region */
  Point     offset;     /* where (0,0) really is */
  CopyRect  copy_rect;  /* pointer to drawing func */
  FillRect  fill_rect;  /* pointer to drawing func */
  DrawUTF8  draw_utf8;  /* pointer to drawing func */
  void *    extra;      /* platform-specific data */
};
```

**FUNCTIONS**

```
Graphics *get_window_graphics(Window *w);
Graphics *get_control_graphics(Control *c);
Graphics *get_bitmap_graphics(Bitmap *b);
Graphics *get_image_graphics(Image *i);

void    del_graphics(Graphics *g);

void    set_rgb(Graphics *g, Colour col);
void    set_rgbindex(Graphics *g, int index);

void    set_xor_mode(Graphics *g, Colour alt);
void    set_paint_mode(Graphics *g);

void    set_line_width(Graphics *g, int width);
void    set_font(Graphics *g, Font *f);

void    set_clip_rect(Graphics *g, Rect r);
void    set_clip_region(Graphics *g, Region *rgn);
```

**NOTES**

A *Graphics* object is required whenever a program needs to draw. A graphics object is an abstract object obtained prior to drawing, and is released after drawing has finished. Drawing functions (see later sections) use a graphics object as a common parameter when drawing to windows, controls, bitmaps and images. Hence, the drawing operations all behave the same on different objects, through the use of the common graphics object interface.

Sometimes the system obtains and releases a graphics object automatically for the program, for example, in certain window system call-back functions. When a window or control is redrawn, its *redraw* call-back function is called by the system. This will be defined as a *WindowDrawFunc* or a *DrawFunc* (see above definitions for details). The first parameter passed to this function is the object being drawn, the second parameter is a graphics object obtained by the system. The programmer writes code in the call-back function to use the graphics object for drawing, but does not need to release the graphics object; the system automatically releases it after the call-back has returned.

If the programmer needs to draw to an object, but does not currently have a valid graphics object to use, one can be obtained using **get_window_graphics** (to draw to a window), **get_control_graphics** (to draw to a control on a window), **get_bitmap_graphics** (to draw to a bitmap) or **get_image_graphics** (to draw to an image in memory).

Graphics objects obtained through one of those functions should later be deleted using **del_graphics**. As stated earlier, this function should not be used on a graphics object passed to a call-back by the system.

A graphics object keeps track of the target object (the object to which drawing is directed) as well as some drawing state information, such as the current drawing colour (which is initially black), the text drawing font (which is initially a Unicode system font) and the pixel width to draw lines (which is initially set to one).

The **set_rgb** function causes the drawing colour to change to the specified colour. There are two synonyms for this function: **set_colour** and **set_color** both do the same thing. Note that it is not correct to simply change the `colour` field in the graphics object directly; the drawing colour must be changed via the **set_rgb** function or nothing will happen. The `colour` field is merely a way of reporting what the system believes is the current drawing colour. If the alpha field in the chosen colour is CLEAR (255), no drawing will occur.

The **set_rgbindex** function changes the drawing colour to a particular integer pixel value, by-passing the sometimes slow colour-matching algorithms used in **set_rgb**. This can only be used on indexed-colour displays; it will not work as expected on TrueColour or DirectColour displays.

The **set_xor_mode** function changes the drawing mode so that the pixels being drawn are combined with existing pixels using a bitwise exclusive-or operation. The `alt` parameter gives the alternate colour with which drawing will occur. When drawing shapes, pixels which are the alternate colour will become the current drawing colour, and vice versa. Interactions with other colours are unpredictable but reversible; if the same shape is drawn twice the original colours will be restored.

The **set_paint_mode** function restores normal drawing operation after the exclusive-or drawing mode has been used. This is the default mode whenever a Graphics object is obtained, and signifies that the drawing colour overwrites existing pixels wherever drawing occurs.

Use **set_line_width** to change the pixel width of lines drawn using the graphics object. The default line width is 1 pixel.

Use **set_font** to change the font used when drawing text using this graphics object. The default font is a Unicode system font. See the section on fonts for details of obtaining fonts.

The **set_clip_rect** function restricts drawing to within a certain rectangle. The rectangle is given in co-ordinates relative to the object to which drawing is directed

(the target object). All drawing will thereafter be clipped to within that rectangle.

The **set_clip_region** functions restricts drawing to a given region (collection of rectangles). The region is given in co-ordinates relative to the target object. All drawing will be clipped to the region, so that no pixels outside that region will be changed. The region is copied by this function, so the original region can be safely modified or deleted after this function has been called, without affecting the clipping region.

Initially, drawing is only clipped to the boundaries of the target object. When setting a new clipping rectangle or region, drawing is also still clipped to the rectangle of the target object. Hence, it is not possible to draw outside a window, control, bitmap or image.

A note about windows: if a graphics object for a window has been obtained and then the window is resized, the graphics object must be destroyed and obtained again. Otherwise the clipping region in the graphics object will be incorrect, and may restrict drawing to the wrong areas.

**EXAMPLES**

- editdraw.c

## 6.2 Drawing Functions

### FUNCTIONS

```
int  draw_point(Graphics *g, Point p);
int  draw_rect(Graphics *g, Rect r);
int  fill_rect(Graphics *g, Rect r);
int  draw_shadow_rect(Graphics *g, Rect r,
                      Colour c1, Colour c2);
int  draw_line(Graphics *g, Point p1, Point p2);
int  draw_round_rect(Graphics *g, Rect r);
int  fill_round_rect(Graphics *g, Rect r);
int  draw_ellipse(Graphics *g, Rect r);
int  fill_ellipse(Graphics *g, Rect r);
int  draw_arc(Graphics *g, Rect r,
              int start_angle, int end_angle);
int  fill_arc(Graphics *g, Rect r,
              int start_angle, int end_angle);
int  draw_polyline(Graphics *g, Point *p, int n);
int  draw_polygon(Graphics *g, Point *p, int n);
int  fill_polygon(Graphics *g, Point *p, int n);
void draw_all(App *app);
```

### NOTES

All drawing operations require a valid *Graphics* object to allow drawing. This graphics object is used to determine where and how pixels will be drawn. See the section on graphics object for details on how to obtain a graphics object to draw to a window, control, bitmap or image.

All of the drawing operations described here return 1 on success, or 0 if a memory error prevents successful completion.

The **draw_point** function sets the colour of the given point to be the current drawing colour. It will change only one pixel, regardless of the current line width.

The **draw_rect** function draws a rectangle within the given rectangle. The lines will have a thickness defined by the current line width and will be wholly within the rectangle.

The **draw_shadow_rect** function is similar, but draws the box in two colours. The first colour `c1` is used to draw the top and left portions of the box, and the second colour `c2` is used to draw the bottom and right portions. This creates a raised border effect, as used by push-buttons and some other controls.

The **fill_rect** function fills the specified rectangle with the current colour.

The **draw_line** function draws a line starting at point `p1` and extending up to but not including point `p2.` The line will have a width equal to the current line width.

The **draw_round_rect** function draws a box with rounded edges. The **fill_round_rect** function fills the rounded box with the current colour.

The **draw_ellipse** function draws a complete ellipse within the supplied rectangle. Remember that the right and bottom edges of a rectangle are not included within the rectangle. To draw a circle, make the rectangle a square. The **fill_ellipse** function fills the corresponding ellipse with the current colour.

The **draw_arc** function draws an ellipsoid arc centred in the middle of the rectangle `r`, extending anti-clockwise from the `start_angle` to the `end_angle.` Angles are measured in degrees, with 0 degrees being in the 3 o'clock position on the arc. The arc will fit within the rectangle. The **fill_arc** function creates a pie-shape with the end-points of the arc joined to the centre point.

To draw many lines at once, the **draw_polyline** function is used. It is passed an array of `n` points, and connects each point to the next in the array using **draw_line**.

The **draw_polygon** function is given an array of `n` points. It will draw lines from the first point in the array to the next, and so on until it joins the last point back to the first. The **fill_polygon** function will create a polygon filled with the current colour.

Graphics operations on some platforms (such as X-Windows) may be buffered, and for those platforms calling **draw_all** ensures all pending graphics requests are processed immediately. On other platforms the function exists and does nothing. This function is called during event handling anyway, and so is not generally called explicitly.

**EXAMPLES**

- arcs.c
- ellipses.c
- polygons.c
- smiley.c

## 6.3 Copying Pixels

### FUNCTIONS

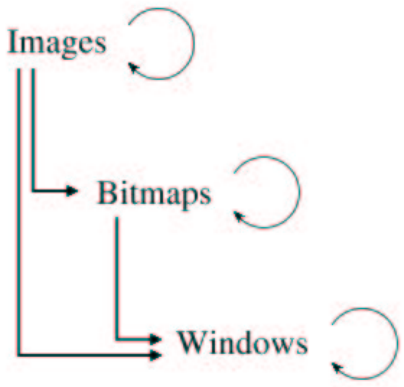
```
int     copy_rect(Graphics *g, Point dp, Graphics *src, Rect sr);
void    texture_rect(Graphics *g, Rect dr, Graphics *src, Rect sr);
void    draw_image(Graphics *g, Rect dr, Image *img, Rect sr);
Bitmap *image_to_bitmap(Window *win, Image *img);
```

### NOTES

The **copy_rect** operation copies pixels from the source rectangle `sr` in the source graphics object `src`, to the destination point `dp` in the destination graphics object `g`. There are some restrictions on this function:

- Copying pixels from bitmaps into images will not work. Bitmaps are output devices.
- Copying pixels from windows into images will not work. Windows are output devices.
- Copying from bitmaps to windows is only guaranteed to work if the bitmap has the same arrangement of colour data as the window (i.e. that window was passed as a parameter to **new_bitmap** when creating the bitmap).
- Copying bitmaps to other bitmaps is only guaranteed to work if the two bitmaps share the same colour data arrangement.
- Copying from a window to a bitmap shares the same restriction, but additionally might produce incorrect pixels if portions of the source window are obscured by another window.
- Copying from a window to another window shares the same restrictions, for the same reasons.
- Copying from a window to itself is an exception; this is guaranteed to work because any portions which are obscured while copying will be refreshed automatically by calling the window's redraw call-back during the next event-handling cycle. Hence, this function can be used to perform window scrolling.

For these reasons, it is best to think of bitmaps and windows as output devices only, with copying going from images to bitmaps to windows, but never in the other direction:




*Valid drawing destinations.*

The **texture_rect** function overlays the entire destination rectangle `dr` with copies of the source rectangle `sr` from the pixel source `src`, starting in the top-left point of `dr` and proceeding to the left and down. This produces a wall-paper effect.

The **draw_image** function specialises in drawing a scaled version of an image. It copies pixels from the source rectangle `sr` of the image `img` into the destination rectangle `dr`, scaling between the two rectangles as required. When drawing an image many times, it is sometimes more efficient to scale the image first then use **copy_rect** instead of **draw_image**, since the scaling is only performed once.

Drawing an image to a bitmap or window is implemented by creating a temporary bitmap, copying the image data into the bitmap, copying the bitmap to the window, then destroying the temporary bitmap. For this reason, when drawing an image many times, it is sometimes more efficient to create a bitmap from the image using **image_to_bitmap**, and then draw from that bitmap instead.

**EXAMPLES**

- imgtest.c

## 6.4 Drawing Text

### FUNCTIONS

```
int   draw_utf8(Graphics *g, Point p, char *utf8, int nbytes);
char *draw_text(Graphics *g, Rect r, int align, char *utf8, int nbytes);
int   text_height(Font *f, int width, char *utf8, int nbytes);
int   text_width(Font *f, int width, char *utf8, int nbytes);
int   text_line_length(Font *f, int width, char *utf8, int nbytes);
void  set_text_direction(Graphics *g, int direction);
```

### CONSTANTS

```
enum {
  ALIGN_LEFT     = 1,
  ALIGN_RIGHT    = 2,
  ALIGN_JUSTIFY  = 3,
  ALIGN_CENTER   = 4,
  ALIGN_CENTRE   = 4,
  VALIGN_TOP     = 8,
  VALIGN_BOTTOM  = 16,
  VALIGN_JUSTIFY = 24,
  VALIGN_CENTER  = 32,
  VALIGN_CENTRE  = 32,
  LEFT_TO_RIGHT  = 64,
  RIGHT_TO_LEFT  = 128
};
```

### NOTES

The **draw_utf8** function draws the Unicode text characters from the UTF-8 encoded string, using the current font and the current colour. The number of bytes in the string is given by `nbytes`, so the string may contain '\0' characters if required.

The upper left corner of the first character is placed at point `p`, and subsequent characters are placed to the right of each previous character. It will *not* wrap the text to the next line if the edge of a window is encountered; it will instead simply stop drawing. The function returns one if it succeeded, or zero if a problem happened, such as an inability to load parts of the font or a lack of memory.

The **draw_text** function draws UTF-8 text within a bounding box, using a given text-alignment. Words are wrapped to the next line as necessary. The possible text-alignments are: `ALIGN_LEFT`, `ALIGN_RIGHT`, `ALIGN_JUSTIFY`, `ALIGN_CENTRE`, `VALIGN_TOP`, `VALIGN_BOTTOM`, `VALIGN_JUSTIFY` or `VALIGN_CENTRE` (American spellings are also allowed). The first four of these refer to the horizontal alignment of text within the bounding box, the other four refer to the vertical placement of text. A horizontal and a vertical alignment may be combined using the plus or bit-wise or operators. An alignment of zero is equivalent to `ALIGN_LEFT+VALIGN_TOP`.

The **draw_text** function draws only as much text as will fit in the bounding box. It uses the current colour and the current font. The function returns a pointer to the first character in the text string which could not be drawn within the bounding box; hence this pointer can be used to continue drawing the text string elsewhere. It will return NULL if the entire string was drawn.

The **text_height** function returns the pixel height of the given string using the given font and a supplied maximum pixel width for the UTF-8 text. The number of lines over which the text will be displayed can thus be determined by dividing the returned pixel height by the height of the font.

The **text_width** function returns the pixel width of a line of UTF-8 text, if it was drawn in the given font. The maximum pixel width parameter specifies how wide a space the text is to be drawn within. Tab characters will align on boundaries which are multiples of 8 spaces, and newline characters will end the line, otherwise the line ends when a word wraps to the next line.

The **text_line_length** function returns the length in bytes of the largest line of text which will fit in the given pixel width in the given font. If a newline occurs in the text, this will end the line and return the length.

The **set_text_direction** function sets the direction in which text will be drawn. The default direction is `LEFT_TO_RIGHT`, but this can be changed to `RIGHT_TO_LEFT`. In that case, the point passed to **draw_utf8** should refer to the *top-right* point of the text to be drawn, not the top-left, and the text will be drawn a character at a time to the left of that point. Similarly, **draw_text** will still draw all text within the rectangle, but will draw the letters and words from the right to the left. Alignment is separate to text direction, so it is possible to have text written right-to-left, but left-aligned.

**EXAMPLES**

- viewutf8.c

# Chapter 7

# Miscellaneous

## 7.1 Clipboard Functions

### FUNCTIONS

```
int    set_clipboard_text(App *app, char *text);
char * get_clipboard_text(App *app);
```

### NOTES

The **set_clipboard_text** function causes the system clipboard to contain the specified zero-terminated text string. It returns zero if it fails for some reason, or non-zero if it succeeds.

The **get_clipboard_text** function retrieves the text string from the system clipboard (if it contains a text string). If the clipboard does not contain text, or it is empty, this function will return NULL.

Text may be modified in the process of saving it to the clipboard. It is not guaranteed that non-ASCII characters, or control characters other than tab and newline, can be transferred using this mechanism.

### EXAMPLES

- char.c

## 7.2 Dialog Boxes

**FUNCTIONS**

```
void  error(App *app, char *message);

void  ask_ok(App *app, char *title, char *message);
int   ask_ok_cancel(App *app, char *title, char *question);
int   ask_yes_no(App *app, char *title, char *question);
int   ask_yes_no_cancel(App *app, char *title, char *question);

char *ask_string(App *app, char *title, char *question,
                      char *default_answer);
char *ask_file_open(App *app, char *title, char *message,
                         char *path);
char *ask_file_save(App *app, char *title, char *message,
                         char *path);
```

**CONSTANTS**

```
enum {
  CANCEL = -1,
  NO     =  0,
  YES    =  1
};
```

**NOTES**

The **error** function displays the `message` string in a modal message window before terminating the application.

The rest of the functions which accept a `title` string parameter, display a modal dialog box with the given `title` string showing in the title bar.

The **ask_ok** function displays the `message` string in a modal window with an "OK" button which the user can select to dismiss the window.

The **ask_ok_cancel** function displays a modal window with a `question`, and returns YES if the user clicks the "OK" button, and CANCEL if the "Cancel" button is used to dismiss the window.

The **ask_yes_no** function displays the `question` string in a modal window with two buttons, "Yes" and "No". The user may select either one to dismiss the

window. If the user selects “Yes”, the function returns YES; if the user selects “No” the function returns NO.

The **ask_yes_no_cancel** function operates the same way except it has another button “Cancel” which, if selected, will cause the function to return CANCEL.

The **ask_string** function displays the `question` string in a modal window with the specified `title`, which also has a text field into which the user can type a string. The text field is initially loaded with the `default_answer`. Space is allocated for the string using **alloc** and then returned to the programmer by the function (use **free** to free the returned string). If the user selects the “Cancel” button in this window, the function will instead return NULL.

The **ask_file_open** function displays a modal window which can be used to select a file from the file system. The `path` string is loaded into the filename text field if this is available. The function returns the complete file pathname as a string, or NULL if the user cancels the operation. The returned string should be freed using **free** when it is no longer needed.

The **ask_file_save** function is similar, except that it is used when saving a file, and the window may appear differently to signal that fact. It will warn the user if a selected file already exists to confirm overwriting the file is allowed. The returned file path string should be freed using **free** when it is no longer needed.

## 7.3 Files and Folders

### FUNCTIONS

```
FILE *open_file(char *filepath, char *mode);
int   close_file(FILE *f);
int   remove_file(char *filepath);
int   rename_file(char *oldpath, char *newpath);

Folder *open_folder(char *foldername);
char *  read_folder(Folder *f);
int     close_folder(Folder *f);
int     make_folder(char *folderpath, int mode);
int     remove_folder(char *folderpath);
int     rename_folder(char *oldpath, char *newpath);
char *  current_folder(void);
int     set_current_folder(char *folderpath);

int   file_info(char *filepath);
long  file_size(char *filepath);
long  file_time(char *filepath);

char *form_file_path(char *foldername, char *filename);
char *get_file_name(char *filepath);
```

### CONSTANTS

```
enum FileInfo {
  IS_APP    = 1,
  IS_WRITE  = 2,
  IS_READ   = 4,
  IS_FOLDER = 8,
  IS_FILE   = 16
};
```

### NOTES

Operating systems often differ when it comes to accessing files and directories (folders). There are many differences, but two of the main differences are the directory separator character (which is '/' in Unix), and the way to up one directory level (which is '..' in Unix).

The functions in this section have been provided to allow all code to access files and folders using the Unix-style notation.

Ordinarily, a standard C function such as `fopen` accepts a platform-specific C string to use to locate a file. On a Linux platform this might be a string such as "../manual/folders.htm" while on Windows this would be "..\manual\folders.htm" and on some Macintosh systems might be written as "::manual:folders.htm". These differences cause difficulties when porting C source code.

The **open_file** function can be used as a cross-platform replacement for **fopen**. It translates the given Unix-style file path string into whatever style the operating system requires. On Windows, for instance, it exchanges '/' for '\' in a copy of the file path string, then calls **fopen**, then frees the temporary string. Using **open_file** instead of **fopen** allows code to assume Unix-style file and directory names everywhere.

Correspondingly, **close_file** replaces **fclose**, except that it returns 1 for success, and 0 for failure. Passing a NULL file pointer to this function is valid and returns 1.

The **remove_file** function attempts to delete the named file, returning 1 for success, 0 for failure. The name is given in Unix-style notation.

The **rename_file** function renames a file on the file system from the old path name to the new, both given in Unix-style notation. This can be used to rename a file, and/or move a file to a new folder. It returns 1 for success, 0 for failure.

The **open_folder** function opens a directory for reading, using a Unix-style directory specifier. The ".." path component causes a textual removal of the previous path component, if any, and "." components are removed, since they represent the same directory. Again, '/' represents the directory path separator on all platforms, but is internally substituted with the platform-specific separator.

The **read_folder** function reads one file or directory name from the open folder, returning a pointer to a static zero-terminated string which contains the name, or NULL if there are no more names listed in that directory. The names "." and ".." will usually be present as the first items on the list. The list of names will not be sorted into alphabetic order.

The **close_folder** function closes an open folder, returning 1 for success, 0 if there is an error. Passing a NULL folder pointer to this function is valid and returns 1.

Use **make_folder** to create a new folder with the given name and mode, if possible. The name is given in Unix-style notation. The mode is a bit-field, consisting of three groups of three permission bits, in the order `rwxrwxrwx` where 'r' means

read access, 'w' means write access, and 'x' means execute or search access. The first (highest) group of bits refers to user-level access, the next to group access, and the last to world access. The mode is often written as an octal constant, such as 0755 (which is `rwxr-xr-x` mode), or 0700 (which is `rwx------` mode, or user access only). The mode may be ignored by some implementations if the operating system cannot support such access modes. The function returns 1 for success, 0 for failure.

The **remove_folder** function attempts to delete the named folder from the file system, returning 1 for success, 0 for failure. The name is given in Unix-style notation. The operation will fail if there are any files or folders contained within the folder which is being deleted.

The **rename_folder** function renames a folder on the file system from the old path name to the new, both given in Unix-style notation. This can be used to rename a folder, and/or move a folder into a new folder. It returns 1 for success, 0 for failure.

Use **current_folder** to discover the current working directory, given using Unix-style notation.

The **set_current_folder** function changes the current working directory to the specified path, given in Unix-style notation. It returns 1 for success, 0 for failure.

The **file_info** function is a cross-platform version of the standard C function **fstat**, which returns some flags yielding information about the given file or directory. The returned integer is a bit-field with the following possible values:

- `IS_APP` is set if the executable flag is present on the file or folder.
- `IS_WRITE` is set if the file or folder can be written to.
- `IS_READ` is set if the file or folder can be read.
- `IS_FOLDER` is set if the path refers to a directory.
- `IS_FILE` is set if the path refers to a file.

The **file_size** function reports the size in bytes of the specified file. The path uses Unix-style notation.

The **file_time** function returns the last modification time of the specified file, represented in seconds since the start of the epoch (the current epoch is from Jan 1, 1970). The path uses Unix-style notation.

The **form_file_path** function joins a Unix-style directory path name and a file name, to produce a new correct Unix-style path name. The string is created using **alloc**, and so should later be freed using **free**. The function inserts a '/' between the path and the file name if necessary.

The **get_file_name** function returns a pointer to the start of just the file name from a full path name, discarding the folder information. Because this is just a pointer to the first character in the file name, the returned string should not be freed.

## 7.4 Internationalisation

### FUNCTIONS

```
void  set_string(App *app, char *key, char *value);
char *get_string(App *app, char *key);
```

### NOTES

The **set_string** function associates a string value with a string key, for use in internationalising programs. Keys and values are case-sensitive and UTF-8 encoded. Once a value is associated with a key, that stored value can be returned by passing the key to **get_string**, otherwise NULL is returned.

The dialog functions use certain keys to determine what message strings to display. Changing the values associated with these keys allows a program to internationalise the dialog functions.

The current set of dialog keys and values are given in the table below.

To create a program which uses dialogs in a language other than English, each of the above values should be translated into the new language, using **set_string** after the *App* structure has been created but before using any dialog functions. For example, a French program might begin:

```
int main(int argc, char *argv[])
{
  App *app = new_app(argc, argv);
  set_string(app, "Yes", "Oui");
  set_string(app, "No", "Non");
  set_string(app, "Cancel", "Annuler");
  set_string(app, "OK", "Oui");
  ...
  ask_yes_no(app, "Quitter?", "Quitter le programme?");
  ...
}
```

Some of the defined strings appear on dialog buttons, some in the title bar of dialog windows, some as messages. Those strings that appear in title bars (the last 4 in the given table) may be impossible to properly internationalise, since some window managers incorrectly display Unicode text, and might only allow ISO Latin 1 or even only ASCII text to be used in title bars.

| Key | Default Value |
|---|---|
| “Yes” | “Yes” |
| “No” | “No” |
| “Cancel” | “Cancel” |
| “OK” | “OK” |
| “Quit” | “Quit” |
| “Create” | “Create” |
| “Overwrite” | “Overwrite” |
| “Error” | “Error” |
| “File:” | “File:” |
| “Files:” | “Files:” |
| “Path:” | “Path:” |
| “Folder:” | “Folder:” |
| “Folders:” | “Folders:” |
| “Create file?” | “That file does not exist. Create the file?” |
| “Already a folder!” | “That file name is already is use as a folder.” |
| “Overwrite file?” | “That file already exists. Overwrite the file?” |
| “Create File” | “Create File” |
| “Error: Create File” | “Error: Create File” |
| “Overwrite File” | “Overwrite File” |
| “Error: Save File” | “Error: Save File” |

*Standard keys and default values used by dialog functions.*

## 7.5 Memory Management

### FUNCTIONS

```
void * alloc(long size);
void * zero_alloc(long size);
void * realloc(void *ptr, long newsize);
void   free(void *ptr);
void   debug_memory(int on);
long   memory_used(void);
```

### NOTES

All memory allocation in the library uses the functions described in this section.

The **alloc** function allocates a block of memory `size` bytes in length. This is usually just a safe wrapper around the standard C function **malloc**, but may perform additional integrity checks. It aborts the program if it runs out of memory. Allocating a zero-sized block is valid and returns NULL.

Use **free** to release from memory any blocks allocated by **alloc**. Never use the standard C function **free** on a block. Passing NULL to **free** is allowed, and does nothing.

The **zero_alloc** function uses **alloc** to create a block of memory, and then uses the standard C function **memset** to set every byte of that block to the value zero. It returns NULL only if **alloc** returned NULL.

The **realloc** function accepts a pointer to a memory block which was previously allocated using **alloc**, and resizes it to the `newsize` (in bytes). It keeps the old contents, or as much as will fit in the new block if making it smaller. The block may be moved around in memory, so a new pointer is returned by this function, and the old pointer becomes invalid. If the function runs out of memory, it aborts the program. The block may be resized to zero bytes, in which case the function frees the block and returns NULL. The pointer passed in may be NULL, in which case it just calls **alloc**. This function is essentially a safe wrapper around the standard C function **realloc**, except that it operates only on blocks produced from **alloc**.

The **debug_memory** function activates or deactivates a memory debugger, which replaces internal calls to **malloc**, **realloc** and **free** with calls to custom, portable, debugging functions. Once activated by passing a non-zero integer to this function, debugging should not be deactivated, since this might cause blocks allocated

using a custom allocator to be deallocated using the standard **free** function, which would cause a crash. The option to switch off memory debugging (by passing zero to **debug_memory**) exists because there are certain controlled conditions where it might be possible to do this.

The **memory_used** function attempts to report how many bytes are currently allocated by these memory functions. This will only be correct if the debugging mode was activated before any other library functions were called. If the debugging mode was never activated, the function reports how many bytes have been allocated through **alloc** only, ignoring changes made through calls to **free** or **realloc**.

## 7.6 Regular Expressions

### FUNCTIONS

```
int  regex_match(char *regexp, char *text);
```

### NOTES

The **regex_match** function performs a regular expression match on the given expression `regexp` and `text`. It returns non-zero (true) if there is a match, or zero (false) if there is no match. Both strings are assumed to be UTF-8 strings.

The regular expression language is very simple:

- “abc” matches the exact string “abc”
- “a*bc” matches “a” followed by zero or more chars, then “bc”
- “abc*” matches any string which starts with “abc”
- “*abc” matches any string which ends with “abc”
- “*abc*” matches any string which contains “abc”
- “a?bc” matches “a” followed by any one UTF-8 char, then “bc”
- “*.jpg” matches any string which ends in “.jpg”

The only special characters are ’*’ and ’?’. There is currently no way to escape these characters. The ’.’, ’’, ’$’, and ’\’ characters all have no special meaning. Strings must match at the start and end of the string unless the ’*’ character is used at the start or end. This scheme is similar to a shell match, rather than `grep`.

## 7.7 Resources

### FUNCTIONS

```
FILE * open_resource(char *file, char *resource, long *length);
int    file_has_resources(char *file);
```

### NOTES

The **open_resource** function attempts to locate a named resource within a given file. A resource is a sub-file, associated with a file using the AddRes tool available from the tools folder. This function returns an open file pointer, pointing to the located resource.

The resource descriptor string can be a regular expression as described in the section on regular expressions. The first matching resource will be used if there are several matching names.

If the resource is found, the return value will be non-NULL, and the length parameter will point to the length in bytes of the resource (which will be zero or greater). If the resource is not found, the return value will be NULL, and the length parameter will point to zero.

The file pointer returned by this function will not signal EOF when the resource has ended. Rather, the length parameter must be used, or the resource must have a way of telling the program that it has ended. For example, an embedded image file may contain a marker meaning “end of image”. For embedded text files, the zero byte which separates one resource from the next may be the only other way to determine this end point.

The **file_has_resources** function returns non-zero if the named file ends with the byte sequence that specifies a resource file, otherwise it returns zero. The byte sequence is `"\nApp Resource File Type 1\n"`.

## 7.8 Sound Functions

### FUNCTIONS

```
void  beep(App *app);     /* make a beep noise */
```

### NOTES

The **beep** function beeps the speaker.

Currently there are no other functions for using sounds.

## 7.9 Timer Functions

### OBJECTS

```
typedef struct Timer  Timer;
typedef void (*TimerAction)(Timer *t);

struct Timer {
  App *       app;            /* associated App */
  int         milliseconds;   /* interval between actions */
  int         remaining;      /* time until action */
  TimerAction action;         /* user-defined action */
  void *      data;           /* user-defined data */
  int         value;          /* user-defined value */
};
```

### FUNCTIONS

```
int     delay(App *app, int milliseconds);
long    current_time(App *app);

Timer * new_timer(App *app, TimerAction action, int milliseconds);
void    del_timer(Timer *t);
```

### NOTES

The **delay** function suspends program operation for the required number of `milliseconds`, returning the number of milliseconds elapsed. The return value may be less than the required milliseconds if the delay was interrupted by an external event. The timing of this function is likely to be inaccurate for very small delays, since operating system activity and hardware interrupts are likely to produce small variations in the actual delay time. This function should be used sparingly.

The **current_time** function returns the approximate time in milliseconds since the program began. This number will overflow if the program is used continuously for more than a month.

A *Timer* is an object which allows a program to have a function repeatedly called at certain intervals. It is created by **new_timer**, which is passed the number of `milliseconds` which should elapse between calls to the given *TimerFunction*.

A *TimerFunction* is a user defined function which is called when the timer activates, and is passed a pointer to the *Timer*.

A timer can be stopped using **del_timer**, which also deletes the timer's data structure from memory.

Timers are inexact and not likely to occur more often than at one tenth of a second intervals.

**EXAMPLES**

- polygons.c
- clock.c
- moire.c