where `Id` is the overlay id (0 means the main plot, and you can use -1 to indicate all), and `symbol` is a function that will be called to draw the symbols on the data point. The parameters passed to this function are the object pointer, overlay id, the center of the symbol (`p->x`,`p->y`), number of data points (`n`) and the preferred symbol size (`w`,`h`). If the plot type corresponding to `Id` is not `POINTS_PLOT` or `LINESPOINTS_XYPLOT`, no symbol will be drawn.

For example, to change the `LINEPOINTS` xyplot to plot a filled small circle instead of the default cross, the following can be used

```
void drawsymbol(FL_OBJECT *ob, int id,
                FL_POINT *p, int n, int w, int h)
{
    int r = (w + h) / 4;
    FL_POINT *ps = p + n;

    for (; p < ps; p++)
       fl_circf(p->x, p->y, r, FL_BLACK);
}

....
fl_set_xyplot_symbol(xyplot, 0, drawsymbol);
...
```

If *Xlib* drawing routine is used, it should use the current active window (`FL_ObjWin(ob)`) and the current gc (`fl_get_gc()`). Take care not to call routines inside draw symbol function that could trigger a redraw of the xyplot (such as `fl_set_object_color()`, `fl_set_xyplot_data()` etc).

To use absolute bounds as opposed to actual bounds in data, use the following routines

```
void fl_set_xyplot_xbounds(FL_OBJECT *ob, double min, double max);

void fl_set_xyplot_ybounds(FL_OBJECT *ob, double min, double max);
```

Data that fall outside of the range will be clipped. To restore autoscaling, use `max==min`. To reverse the axes (e.g., min at right and max at left), set `min > max` for that axis.

To get the current bounds, use the following routines

```
void fl_get_xyplot_xbounds(FL_OBJECT *ob, float *min, float *max);

void fl_get_xyplot_ybounds(FL_OBJECT *ob, float *min, float *max);
```

Note that the bounds returned are the bounds used in clipping the data, which are not necessarily the bounds used in computing the world/screen mapping due to tic rounding.

To replace the value of a particular point use the routine

```
void fl_replace_xyplot_point(FL_OBJECT *obj,int i,double x,double y)
```

Here index is the index of the value to be replaced. The first value has the index of 0.

It is possible to overlay several plots together using the following call

```
void fl_add_xyplot_overlay(FL_OBJECT *obj, int ID, float *x, float *y,
                          int npoints, FL_COLOR col)
```

where ID must be between 1 and FL_MAX_XYPLOTOVERLAY (32) inclusive. Again, the data is copied internally (old data freed if any)

Similar to the base data, a data file can be used to specify the (x,y) function

```
int fl_add_xyplot_overlay_file(FL_OBJECT *obj, const char *file
                               FL_COLOR col)
```

The function returns the number of data points succesfully read.

The type (FL_NORMAL_XYPLOT etc.) used in overlay plot is the same as the object itself. To change an overlay style, use the following call

```
void fl_set_xyplot_overlay_type(FL_OBJECT *obj, int ID, int type)
```

Note that although the API of adding an overlay is similar to adding an object, an xyplot overlay is not a separate object. It is simply a property of an xyplot object.

To get the data of an overlay, use the following routine

```
void fl_get_xyplot_overlay_data(FL_OBJECT *ob, int ID,
                                float x[], float y[], int *n);
```

where ID specifies the overlay number between 1 and FL_MAX_XYPLOTOVERLAY or the number set via fl_set_xyplot_maxoverlay()(See below). (actually when the ID is zero, this function returns the base data). The caller must supply the storage space for the data. Upon function return, n will be set to the number of data points retrieved.

If needed, the maximum number of overlays an object can have (which by default is 32) can be changed using the following routine

```
int fl_set_xyplot_maxoverlays(FL_OBJECT *ob, int maxoverlays)
```

The function returns the previous maximum number of overlays.

To obtain the number of data points, use the following routine

```
int fl_get_xyplot_numdata(FL_OBJECT *ob, int ID)
```

where ID is the overlay ID with 0 being the base.

To delete an overlay, use the following routine

```
void fl_delete_xyplot_overlay(FL_OBJECT *obj, int ID)
```

It is possible to place inset texts on an xyplot using the following routine (up to `FL_MAX_XYPLOTOVERLAY`, or the value set via `fl_set_xyplot_maxoverlays`, of such insets can be accommodated):

```
void fl_add_xyplot_text(FL_OBJECT *obj, double x, double y,
                        const char *text, int align, FL_COLOR col);
```

where `x` and `y` are the coordinates where text is to be placed and `align` specifies the placement options relative to the specified point (See `fl_set_object_lalign()` for valid options). For example, if you specify `FL_ALIGN_LEFT`, the text will appear on the left of the point and flushed toward the point (See Fig. 21.1. This is mostly consistent with the label alignment except that now the bounding box (of the point) is of zero dimension. Normal text interpretation applies, i.e., if `text` starts with `@`, a symbol is drawn.

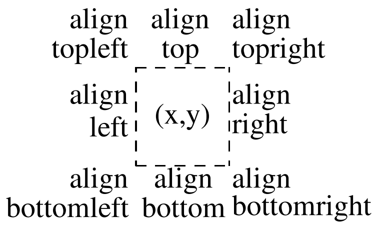

Figure 21.1: Alignment relative to a point

To remove an inset text, use the following routine

```
void fl_delete_xyplot_text(FL_OBJECT *obj, const char *text);
```

Another kind of inset is the keys to the plots. A key is the combination of a segment of the plot line style with a piece of text. Obviously key is useful only when you have more than one plots (i.e., overlays). To add a key to a particular plot, use the following routine

```
void fl_set_xyplot_key(FL_OBJECT *ob, int id,  const char *keys)
```

where `id` again is the overlay ID. To remove a key, set the key to null.

All the keys will be drawn together inside a box. The position of the keys can be set via the following routine

```
void fl_set_xyplot_key_position(FL_OBJECT *ob, float x, float y,
                                int align)
```

where `x` and `y` should be given in world coordinate system. `align` specifies the alignment of the entire key box relative to the given position (See Fig.21.1).

The following routine combines the above two functions and may be more convenient to use

```
void fl_set_xyplot_keys(FL_OBJECT *ob,char *keys[],float x,float y,
                        int align)
```

where `keys` specifies the keys for each plot. The last entry should be null to signify the end. The array index is the plot id, i.e., `key[0]` is the key for the base plot, `key[1]` is the first overlay etc.

To change the font the key text uses, the following routine is available

```
void fl_set_xyplot_key_font(FL_OBJECT *ob, int style, int size)
```

Data may be interpolated using nth order Lagrangian polynomial

```
void fl_set_xyplot_interpolate(FL_OBJECT *ob, int ID, int degree,
                               double grid)
```

where `ID` is the overlay ID (use 0 for the original data) of the xyplot; `degree` is the order of the polynomial to use and `grid` is the working grid onto which the data are to be interpolated. To restore the default linear interpolation, use `degree` 0 or 1.

To change the line thickness of an xyplot (base or overlay), the follow routine is available

```
void fl_set_xyplot_linewidth(FL_OBJECT *ob, int ID, int width)
```

Again, use `ID` 0 to indicate the base data. Setting `width` to zero restores the server default and typically is the fastest.

By default, linear scale in both the X and Y direction is used. To change the scaling, use the following call

```
void fl_set_xyplot_xscale(FL_OBJECT *ob, int scale, double base)

void fl_set_xyplot_yscale(FL_OBJECT *ob, int scale, double base)
```

where the valid scaling options for `scale` are `FL_LINEAR` and `FL_LOG`, and `base` is used only for `FL_LOG` and in that case it is the base of the log desired.

Use the following routine to clear an xyplot

```
void fl_clear_xyplot(FL_OBJECT *ob)
```

This routine frees all data associated with an xyplot, including all overlays and all inset text. This routine does not reset all plotting options, such as line thickness, major/minor divisions etc nor does it free *all* memories associated with the xyplot, which `fl_free_object()` does.

The mapping between the screen coordinates and data can be obtained using the following routines

```
void fl_get_xyplot_xmapping(FL_OBJECT *ob, float *a, float *b)

void fl_get_xyplot_xmapping(FL_OBJECT *ob, float *a, float *b)
```

where a and b are the mapping constants and are used as follows

$$screenCoord = a \times data + b \qquad (linear scale)$$

$$screenCoord = a \times \log_p(data) + b \qquad (log scale)$$

where p is the base[1].

If you need to do conversions only occasionally (for example, converting the position of a mouse click to a data point or vice versa) the following routines might be more convenient

```
void fl_xyplot_s2w(FL_OBJECT *ob, double sx, double sy,
                   float *wx, float *wy);

void fl_xyplot_w2s(FL_OBJECT *ob, double wx, double wy,
                   float *sx, float *sy);
```

where sx and sy are the screen coordinates and wx and wy are the world coordinates.

Remarks

Don't use FL_NO_BOX for an xyplot object that is to be changed dynamically.

To change the font size and style for the tic labels, inset text etc., use fl_set_object_lsize() and fl_set_object_lstyle().

The interpolation routine is public and can be used in the application program

```
int fl_interpolate(const float *inx, const float *iny, int num_in,
                   float *outx, float *outy, double grid, int ndeg)
```

If successful, the function returns the number of points in the interpolated function ((inx[num_in-1] - inx[0]) / grid + 1.01) else it returns -1. Upon return, x and outy are set to the interplated values. The caller should allocate the storage for outx and outy.

Color1 controls the color of the box and Color2 controls the actual xyplot color.

See xyplotall.c and xyplotactive.c for examples of the use of xyplot objects. There is also an example xyplotover.c showing the use of overlay. In addition, xyplotall.c shows a way of getting all mouse clicks without using active xyplot.

It is possible to generate a POSTSCRIPT output of the xyplot. See fl_object_ps_dump() documented in Part V.

[1] $\log_p(x)$ can be computed as $\log_{10}(x)/\log_{10}(p)$ using the math library routine log10(x)

21.3 Pop-ups

XPopup is not really an object class, but because it is used by `FL_MENU` and `FL_CHOICE`, and can function stand-alone, it is documented here.

Short description

XPopups (XPups) are simple transient windows that show a number of choices the user can click on to select the desired options.

Define a new popup

To define a new popup, use the following routines

```
int fl_newpup(Window parent);

int fl_defpup(Window parent, const char *str, [, args ...]);
```

Both functions allocate and initialize a new popup menu and return the menu identifier (-1 if error). `fl_defpup()` in addition accepts a pointer to the text you want to add as a menu item. More than one item can be specified by using a vertical bar (`|`) between the items, e.g., `"foo|bar"` adds two menu items. The `parent` parameter specifies the window to which the pup belongs. In a situation where pup is used inside an object callback (e.g., `FL_MENU_BUTTON`), `FL_ObjWin(ob)` would suffice.

It is possible to pair an “item type" flag with each menu item to specify particular attributes of the item, such as shortcuts and callbacks, etc. If an item requires an argument because of the type, the argument must be supplied by the variable arguments `arg`.

The following menu types are supported (to get a normal %, stack two together just like in `printf`):

`%t` Marks item text as the menu title string.

`%F` Invokes a routine for every selection made from this menu. You must specify the invoked routine in the `arg` parameter. The value of the menu item is used as a parameter of the executed routine. Thus if you select the third menu item, the system passes 3 as a parameter to the function specified by `%F`.

`%f` Invokes a routine when this particular item is selected. The routine must be supplied in the `arg` parameter. The value of the menu item is passed as a parameter of the routine. If you have also bound the entire menu to a callback function via `%F`, then the result of the `%f` routine is passed as a parameter of the `%F` routine.

`%d` Disables and gray-out this item.

`%i` Makes this item inactive.

`%l` Adds a line under the current entry. This is useful in providing visual clues to group like entries together.

`%m` Whenever this menu item is selected, pop up another menu (cascade menus). The new menu identifier must be provided in the `args` argument.

`%h` Specify "hotkeys" that can be used to select this item. Hotkeys must be given in the `args` parameter as a pointer to string. Use `#` to specify <ALT>, `^` <CONTROL>, and `&n` key `Fn`.

`%xn` Assigns a numeric value to this menu item. This value must be positive. This new value overrides the default position-based value assigned to this item. Different from other flags, the value `n` must be entered as part of the text string. Do not use the `arg` parameter to specify numeric value.

`%b` Indicates this menu item is binary (toggle). When displayed, binary item will be drawn with a small box to the left. See also `FL_PUP_BOX`.

`%B` Same as `%b` except it also signifies this item is "true" and consequently the item is drawn with a checked box on the left. See also `FL_PUP_CHECK`.

`%rg` Specifies this menu item is a "radio item" belonging to group `g`. Group number `g` must be positive and within (1-64). A radio group is drawn with a small diamond box to the left.

`%Rg` Same as `%rg` except that it also sets the state of the radio item as "pushed", i.e., the item is drawn with a diamond box to the left. See also `fl_setpup_selection()`.

`cntl-H (010)` Same as `%l` except that the character must precede the item label, i.e., use `"\010Abc"` rather than `"Abc\010"`.

Due to variable arguments, error checking is minimal. In addition, if `%x` is used to specify a value that happens to be identical to a position-based value, the result is unpredictable in subsequent reference to these items. Also there is currently a limit of `FL_MAXPUPI`(64) items per popup.

Tabs (`\t`) can be embedded in the item string to align different fields.

You can add more menu items to an existing popup menu using the following routine

```
void fl_addtopup(int menuID, const char *str, ...);
```

Again, `str` can contain the special sequences mentioned earlier.

To display a popup, use the following routine

```
int fl_dopup(int menuID);
```

This function displays the specified popup menu until the user makes a selection. The value returned is the value of the item selected. However, if there are functions bound to the menu or menu item, the function is invoked with the value as a parameter and the value returned by `fl_dopup` is the executed function value. If no selection is made, function returns -1. Selecting the title box or invoking the pop-up via a non-pointer event results in a "hanging" pop-up, and you can re-select or choose to navigate using the keyboard.

A typical procedure may look as follows:

```
int item3_cb(int n) { /* handle this */ return whatever; }

/* define the menu */
int menu = fl_newpup(parent);
fl_addtopup(menu,"Title %t|Item1|Item2|Item3%x10%f|Item4",item3_cb);

switch(fl_dopup(menu))
{
   case 1:   /* item1 is selected */
     /* handle it */
   case 2:
     /* handle it */
   case 4:
     /* handle it */
   case whatever:
    /* item 3 call back has been executed */
}
```

Since `item3_cb` is bound to item3, upon whose selection, instead of returning 10, the bound function is called with 10 as the parameter, i.e., `item3_cb(10)`. The value returned by `item_cb(10)` is returned by `fl_dopup()`.

Sometimes, it might be necessary to obtain the popup ID (for example, inside an item callback function). To this end, the following function available:

```
int fl_current_pup(void)
```

If no popup is active, the function returns -1.

To destroy a popup menu and release all memory used, use the following routine

```
void fl_freepup(int menuID);
```

For most applications, the following simplified API may be easier to use

```
void fl_setpup_entries(int ID, FL_PUP_ENTRIES *entries)
```

where `Id` is the popup ID returned by `fl_newpup()` or `fl_defpup()` and `entries` is an array of the following structures

```
typedef struct
{
   const char *item_text;     /* item text label            */
   FL_PUP_CB callback;        /* item callback routine      */
   const char *shortcut;      /* shortcut for this item     */
   unsigned int mode;         /* item mode                  */
} FL_PUP_ENTRY;
```

The meaning of each member of the structure is as follows

`text` This is the text of a popup item. If `text` is null, it signifies the end of this popup menu. The first letter of the text string has special meaning if it is one of the following:

['/'] This indicates the beginning of a submenu from the next item through next null.

['_'] Indicates that an underline should be drawn under this item.

`callback` This is the callback function that will be called when this particular item is selected by the user. `fl_dopup()` returns the value returned by this callback. If the callback is null, the item number will be returned by `fl_dopup()`.

`shortcut` Specifies the keyboard shortcut.

`mode` Specifies the attributes (`FL_PUP_GRAY` etc) of this item.

With this simplified API, each popup item is assigned a value (via %x) automatically. The item value starts from 1 and is the corresponding index in `entries` array plus 1. For example, the third entry in the structure has a value of 4. This way, the application can relate the value returned by `fl_dopup()` to the array easily. See demo program `popup.c` for an example use of the API.

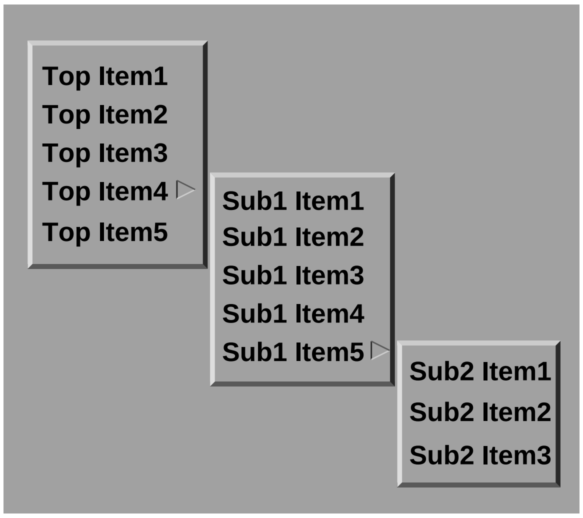

Figure 21.2: An example of a popup menu

To illustrate the usage of `fl_setpup_entries()`, Fig 21.2 shows the popup created by the structure defined in the following code segment.

```
FL_PUP_ENTRY entries[] =
{
    {"Top item1", callback},        /* item number 1 */
    {"Top item2", callback},
    {"Top item3", callback},
```

```
      {"/Top item4", callback},
         {"Sub1 item1", callback},   /* item number 5 */
         {"Sub1 item2", callback},
         {"Sub1 item3", callback},
         {"Sub1 item4", callback},
         {"/Sub1 item5", callback},
            {"Sub2 item1", callback}, /* item number 10 */
            {"Sub2 item2", callback},
            {"Sub2 item3", callback},
            {0},                     /* end of level2. Item number 12*/
 {0},                        /* end of sub level1 */
      {"Top item5", callback},
      {0}                            /* end of popup       */
  };
```

Interaction

To select an item, drag the mouse to the item to be selected and release the mouse. If the position prior to releasing is within the title bar, a “hanging" pop-up results. You can re-select by clicking on or dragging to the item to be selected.

It is possible to use the keyboard to navigate the popup. Specifically use ↑ and ↓ to change the currently marked item; use <RETURN> to select. <ESC> cancels the selection, causing -1 being returned. <HOME> and <END> selects, respectively, the first and the last item.

It is also possible to use convenience functions to bind keyboard keys to menu items (the “hotkeys") instead of `%s`:

```
void fl_setpup_shortcut(int menuID, int item_val, const char *hotkeys)
```

where `item_val` is the value `fl_dopup()` would have returned if that item was selected (%x or position) and `hotkeys` is a string specifying all the hotkey combinations. See Section 24.1 for details. Briefly, `#` and `^` denote respectively the <ALT> and the <CONTROL> key. `&n` where n=1,2, etc., can be used to denote the function key `n`. Thus if `hotkeys` is set to `"#a^A`, both <CNTRL> `A` and <ALT> `A` are bound to the item. One additional property of the hotkey is the underlining of corresponding letters in the item string. Again, only the first alphabet in the hotkey is used. Therefore, for item string `"A Choice"`, hotkey `"Cc"`, `"#C"` or `"^C"` will result in the `C` in `"A Choice"` being underlined while `"cC"` and `"#c"` will not. There is a limit of maximum 8 shortcut keys.

Two convenience functions are available to set the callback functions for items and menus:

```
typedef int (*FL_PUP_CB)(int);
FL_PUP_CB fl_setpup_itemcb(int menuID, int item_val, FL_PUP_CB cb);
FL_PUP_CB fl_setpup_menucb(int menuID, FL_PUP_CB cb);
```

Similar function exists to set the item enter/leave callback

```
typedef void (*FL_PUP_ENTERCB)(int item, void *data);
typedef void (*FL_PUP_LEAVECB)(int item, void *data);

FL_PUP_ENTERCB fl_setpup_entercb(int menuID,
                                 FL_PUP_ENTERCB cb, void *data)

FL_PUP_LEAVECB fl_setpup_leavecb(int menuID,
                                 FL_PUP_LEAVECB cb, void *data)
```

The function `cb` will be called when the mouse enters or leaves an item on menu `menuID`. Two parameters are passed to the callback function. The first parameter is the item number enter/leave applies and the second parameter is the `data` pointer. To remove an enter/leave callback, use the a null callback.

There is also a function to associate a menu item with a submenu

```
void  fl_setpup_submenu(int menuID, int item_val, int submenuID);
```

Other routines

Note most of the setpup/getpup routines are recursive in nature and the function will search the menu and its all submenus for the item.

It is possible to modify the display characteristics of a given popup menu entry after its creation using the following routine

```
void fl_setpup_mode(int menuID, int item_num, unsigned mode);
```

The following modes (and bitwise ORing thereof) are available

`FL_PUP_NONE` No special characteristics. The default.

`FL_PUP_GREY` Entry is grayed-out and disabled. Selecting a grayed-out entry results in -1 being returned.

`FL_PUP_BOX` Entry has an empty box to the left.

`FL_PUP_CHECK` Entry has a box to the left.

`FL_PUP_RADIO` Radio item, drawn with a box to the left.

Note radio item is drawn with a diamond box to the left while regular binary item is drawn with a square box to the left.

Radio attribute set with `FL_PUP_RADIO` will have a unique and same group ID allocated internally by the popup if the item does not already belong to another radio group.

To obtain the mode of a particular menu item, use the following routine

```
unsigned int fl_getpup_mode(int menuID, int item_num)
```

where `menuId` is the ID returned by `fl_newpup()` or `fl_defpup()` and `item_num` is the value of the item. Note that `item_num` can be an item in one of the submenus of `menuID`.

This comes in handy to check if a toggle or radio item is set

```
if(fl_getpup_mode(menuID, n) & FL_PUP_CHECK)
   item is set
```

There exists also a routine that can be used to obtain the menu item text

```
const char *fl_getpup_text(int menuID, int item_num);
```

In some situations, especially when the popup is activated by non-pointer events (e.g., as a result of an object shortcut), the default placement of popups based on mouse location might not be adequate or appropriate, thus XPup provides the following routine to override the default placement

```
void fl_setpup_position(int x, int y)
```

where `x,y` specifies the location where the top-left corner of the popup should be. `x, y` should be given in screen coordinates (i.e., relative to the root window) with the origin at the top-left corner of the screen. This routine should be used immediately before invoking `fl_dopup()` and the position is not remembered afterwards.

If `x` or `y` is negative, the absolute value is taken to mean the desired location of the right or bottom corner of the popup.

A radio group can be initialized by `%R` or reset programmatically using the following routine

```
void fl_setpup_selection(int menuID, int item_val);
```

The difference is that this routine, in addition to setting the "pushed" state of the item, also resets any previously selected item to an unpushed state. This routine can be used anytime a menuID is active, although there is rarely any need to use this routine as XPup keeps track of the current selection and draws the item accordingly once it is active.

To obtain the number of items in a popup, use the following routine

```
int fl_getpup_items(int menuID)
```

Attributes

Use the following routines to modify the default popup font style, font size and border width:

```
int fl_setpup_default_fontsize(int size);

int fl_setpup_default_fontstyle(int style);

int fl_setpup_default_bw(int bw);
```

The functions return the old size and style respectively.

All pups by default use a right arrow cursor. To change the default cursor, use the following routine

```
Cursor fl_setpup_default_cursor(int cursor)
```

where varcursor is one of the standard cursor names defined in `<X11/cursorfonts.h>`, such as `XC_watch` etc. The function returns the current cursor.

To change the cursor of a particular popup, use the following routine

```
Cursor fl_setpup_cursor(int menuID, int cursor);
```

For example, after the following sequence,

```
m_id = fl_defpup(win, "item1|item2");
fl_setpup_cursor(m_id, XC_hand2);
```

the popup `m_id` will use a "hand" instead of the default arrow cursor.

The appearance of popups (and their associated sub-pups) can be change by the following routines

```
void fl_setpup_shadow(int menuID, int yes);
void fl_setpup_softedge(int menuID, int yes);
void fl_setpup_bw(int menuID, int bw);
```

`FL_PULLDOWN_MENU` by default does not have shadow and has a "softer" look. Note by using a negative value for the border width, the popup automatically becomes "softedge".

The background color and text color of a popup can be changed using the following routine

```
void fl_setpup_default_color(FL_COLOR bkcolor, FL_COLOR textcolor)
```

By default, `bkcolor` is `FL_COL1` and `textcolor` is `FL_BLACK`.

For item that has check box associated with it, the checked color (the default is blue) can be changed with the following routine

```
void fl_setpup_default_checkcolor(FL_COLOR checkcolor)
```

There is by default a limit of 32 popups per process. To enlarge the number of popups allowed, use the following routine

```
int fl_setpup_maxpups(int new_max)
```

The function returns the current limit.

It is possible to use popups as a message facility using the following routines

```
void fl_showpup(int menuID)

void fl_hidepup(int menuID)
```

No interaction takes place with a pup shown by `fl_showpup` and can only be removed from the screen programmatically via `fl_hidepup`.

Too additional routines are available that might be useful for moving popups around:

```
void fl_reparent_pup(int MenuID, Window newparent)

void fl_getpup_window(int MenuID, Window *parent, Window *win)
```

Note however, the pup window itself might not get created before `fl_dopup()`. The first routine can be used to change the parent of the popup even if the pup window itself is not created yet.

Remarks

Take care to make sure all items, including the items on submenus, have unique values and are positive.

Pop-ups are used indirectly in `menu.c`, `boxtype.c` and others. For a direct pop-up demo, See `pup.c`. All these programs are located in the `DEMOS/` directory.

21.4 Canvas

Scrolled canvas is not functional yet.

Short description

A canvas is a managed plain X (sub)window. It it different from the free object in that a canvas is guaranteed to be associated with a window that is not shared with any other object, thus an application program has more freedom in utilizing a canvas, such as using its own colormap or rendering double-buffered OpenGL in it etc. A canvas is also different from a raw application window because a canvas is decorated differently and its geometry is managed, e.g., you can use `fl_set_object_resize()` to control its position and size after its parent form is resized

Adding an object

To add a canvas to a form you use the routine

```
FL_OBJECT *fl_add_canvas(int type, FL_Coord x, FL_Coord y,
                        FL_Coord w, FL_Coord h, const char *label)
```

The meaning of the parameters is as usual. The `label` is not drawn but used as the window name for possible resource and playback purposes. If label is empty, window name will be generated on the fly as `flcanvas`n, where $n = 0, 1, \ldots$.

Types

The following types of canvases exist:

`FL_NORMAL_CANVAS`	Simple window.
`FL_SCROLLED_CANVAS`	Two scrollbars are added

A scrolled canvas is just a normal canvas with two scrollbars added. A user can set the value for the scrollbar to scroll the entire window or to get signaled when the scrollbar is interactively changed.

Interaction

Canvas is designed to maximize the user's ability to deal with situations where standard form classes may not be flexible enough. With canvases, the user has complete control over everything that can happen to a window. By default, the only event a canvas will receive is `Expose`. To receive other events, the application program has to select them via `fl_add_selected_xevent()`. or `XSelectInput()` or by adding a canvas handler.

The interaction with a canvas is typically set up as follows. First, you register the events you're interested in and their handlers using the following routine

```
typedef int (*FL_HANDLE_CANVAS)(FL_OBJECT *ob, Window win,
                                int win_width, int win_height,
                                XEvent *xev, void *user_data)

void fl_add_canvas_handler(FL_OBJECT *ob, int event,
                        FL_HANDLE_CANVAS handler, void *user_data);
```

Where `event` is the XEvent type, `Expose` etc.

The `fl_add_canvas_handler()` function first registers a procedure with the event dispatching system of the **Forms Library**, then it figures out the event masks corresponding to the event

event and invokes fl_addto_selected_xevent() to solicit the event from the server. Other book keeping (e.g., drawing the box that encloses the canvas, etc.) is done by the object handler.

Since translation from X event to X event mask is not unique, depending on applications, the default translation of X event to event mask by the canvas may or may not match exactly the intention of the application. Two events, namely MotionNotify and ButtonPress, are likely candidates that need further clarification from the application. By default, when a mouse motion handler (MotionNotify) is registered, it is assumed that the application is interested in mouse movements but not in a continous motion monitoring fashion (tracking). If this is not the case and in fact the aplication wants to use the mouse motion as some type of graphics control, the default behavior would appear “jerky" as not every mouse motion is reported. To change the default behavior so every mouse motion is reported, you need to call fl_remove_selected_xevent() with mask PointerMotionHintMask. Further, the mouse motion is reported regardless if the mouse button is pressed or not. If the application is interested in mouse motion only when the mouse button is pressed, fl_remove_selected_xevent() should be called with mask PointerMotionMask|PointerMotionHintMask. With ButtonPress, you need to call fl_addto_selected_xevent() with mask OwnerGrabButtonMask if you are to add or remove other canvas handlers in the button press handler.

To remove a registered handler, use the following routine

```
void fl_remove_canvas_handler(FL_OBJECT *ob, int event,
                              FL_CANVAS_HANDLER handler)
```

After this function call, the canvas ceases to receive the event registered.

To obtain the window ID of a canvas, use the following routine

```
Window fl_get_canvas_id(FL_OBJECT *ob)
```

or use the generic function(macro) (recommended)

```
Window FL_ObjWin(FL_OBJECT *ob)
```

Of course, window ID has meaning only after the form/canvas is shown.

When the canvas or the form the canvas is on is hidden (via fl_hide_object(), fl_hide_form()), the canvas window is destroyed. If the canvas is shown again, a new window ID for the canvas is created. Thus recording the canvas window ID in a static variable is not the right thing to do. It is much safer (and it doesn't add any run-time overhead) to obtain the canvas window ID via FL_ObjWin(). If your application must show and hide the canvas/form repeatedly, you might consider “unmap" the window, a way of removing the window from the screen without actually destroying it and later re-map the window to show it. The Xlib API for doing these are XUnmapWindow(fl_get_display(), win) and XMapWindow(fl_get_display(), win), where win can be form->window->or FL_ObjWin(canvas) depending on if the form or the canvas is being hidden and shown.

Other routines

Upon canvas's creation, all its window related attributes, e.g., visual, depth and colormap, etc. are inherited from its parent (i.e., the form the canvas is on). To modify any attributes of the canvas, use the following routine

```
void fl_set_canvas_attributes(FL_OBJECT *ob, unsigned mask,
                              XSetWindowAttributes *xswa);
```

See *XSetWindowAttributes*(3X) for the definition of the structure members. Note that this routine should not be used to manipulate events.

Other functions exists that can be used to modify the color/visual property of a canvas:

```
void fl_set_canvas_colormap(FL_OBJECT *ob, Colormap map)

Colormap fl_get_canvas_colormap(FL_OBJECT *ob)

void fl_set_canvas_visual(FL_OBJECT *ob, Visual *vi)

void fl_set_canvas_depth(FL_OBJECT *ob, int depth)

int fl_get_canvas_depth(FL_OBJECT *ob)
```

Note that changing visual or depth does not generally make sense once the canvas window is created (which happens when the parent form is shown). Also, typically if you change the canvas visual, you probably should also change the canvas depth to match the visual.

One caution about `fl_set_canvas_colormap()`: when the canvas window goes away, e.g., as a result of `fl_hide_canvas()` or `fl_hide_form()`, the colormap associated with the canvas is freed (destroyed). This likely will cause problems if a single colormap is used for multiple canvases as each canvase will attemp to free the same colormap, resulting in an X error. If your application works this way, i.e., the same colormap is used on multiple canvases (via `fl_set_canvas_colormap()`), you should use the following routine to prevent the canvas from freeing the colormap:

```
void fl_share_canvas_colormap(FL_OBJECT *ob, Colormap colormap)
```

This function works the same as `fl_set_canvas_colormap()` except that it also sets a intenal flag so the colormap is left alone and unfreed when the canvas goes away.

By default, canvases are decorated with an `FL_DOWN_FRAME`. To change the decoration, change the the boxtype of the canvas and the boxtype will be translated into a frame that best approximate the appearance of the request boxtype (e.g., a `DOWN_BOX` is translated into a `DOWN_FRAME` etc). Note that not all frame types are appropriate for decorations.

The following routine is provided to facilitate the creation of a colormap appropriate for a given visual to be used with a canvas:

```
Colormap fl_create_colormap(XVisualInfo *xvinfo, int n_colors)
```

where `n_colors` indicates how many colors in the newly created colormap should be filled with **XForms**'s default colors (to avoid flashing). Note however, the colormap entry 0 is allocated with either black or white even if you specify 0 for `n_color`. To prevent this from happening (so you have a completely empty colormap), set `n_colors` to -1.

By default, objects with shortcuts appearing on the same form as the canvas will “steal" keyboard inputs if they match the shortcuts. To disable this feature, use the following routine with a false `flag`

```
void fl_canvas_yield_to_shortcut(FL_OBJECT *ob, int flag)
```

Attributes

Some of the attributes, such as boxtype, do not apply to the canvas class. col1 of the object, if set, specifies the background of the canvas. By default, a canvas has no background. Col2 controls the decoration color (if applicable).

OpenGL canvases

Derive specialized canvases from the general canvas object is possible. See next subsection for general approaches how this is done. The following routines work for OpenGL (under X) as well as Mesa, [2], a free OpenGL clone.

To add an OpenGL canvas to a form, use the following routine

```
FL_OBJECT *fl_add_glcanvas(int type, FL_Coord x, FL_Coord y,
                           FL_Coord w, FL_Coord h, const char *label)
```

where `type` is the same as the generic canvas.

A glcanvas so created will have the following attributes by default

```
GLX_RGBA, GLX_DEPTH_SIZE,1,
GLX_RED_SIZE,1, GLX_GREEN_SIZE,1, GLX_BLUE_SIZE,1,
GLX_DOUBLEBUFFER
```

The application program can modify these defaults using the following routine (before the creation of glcanvases)

```
void fl_set_glcanvas_defaults(const int *attributes)
```

See *glXChooseVisual*(3G) for a list of valid attributes.

To get the current defaults, use the following routine

[2] It can be obtained from ftp://iris.ssec.wisc.edu/pub/Mesa

```
void fl_get_glcanvas_defaults(int *attributes)
```

It is also possible to change the attributes on a canvas by canvas basis by utilizing the following routine

```
void fl_set_glcanvas_attributes(FL_OBJECT *ob, const int *attributes)
```

Note this routine can be used to change a glcanvas attributes on the fly even if the canvas is already visible and active.

To obtain the attributes of a particular canvas, use the following routine

```
void fl_get_glcanvas_attributes(FL_OBJECT *ob, int attributes[])
```

The caller must supply the space for the attribute values.

To obtain the the glx context (for whatever purposes), use the following routine

```
GLXContext fl_get_glcanvas_context(FL_OBJECT *ob);
```

Note by default, the rendering context created by a glcanvas uses direct rendering (i.e., by-passing the X server). To change this default, i.e., always render through the X server, use the following routine

```
void fl_set_glcanvas_direct(FL_OBJECT *ob, int flag);
```

Remember that OpenGL drawing routines always draw into the window the current context bound to. For application with a single canvas, this is not a problem. In case of multiple canvases, the canvas driver takes care of setting the proper context before invoking the expose handler. In some cases, the application may want to draw into canvases actively. In this case, explicit drawing context switching may be required. To this end, use the following routine

```
void fl_activate_glcanvas(FL_OBJECT *ob)
```

before drawing into glcanvas `ob`.

Finally there is a routine that can be used to obtain the XVisual information that is used to create the context

```
XVisualInfo *fl_get_glcanvas_xvisualinfo(FL_OBJECT *ob);
```

See demo program `DEMOS/gl.c` for an example use of glcanvases.

Remarks

The OpenGL canvas routines documented above are derived from the generic canvas by utilizing some of services provided by the generic canvas. The following is not meant to be an exact documentation of how the OpenGL canvas is implemented, rather it outlines the general steps and approaches needed to create specialized canvases. The actual implementation of the OpenGL canvas is in `FORMS/gl.c`.

All specialized canvases are created by creating a generic canvas first

```
FL_OBJECT *fl_create_canvas(int type, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h, const char *label)
```

A canvas so created has all the properties of a real canvas and you can add it to a form and use it with the event handling routines mentioned earlier. In addition, hooks are provided so additional tasks can be performed before and after the creation of the canvas window:

```
typedef int (*FL_MODIFY_CANVAS_PROP)(FL_OBJECT *);
void fl_modify_canvas_prop(FL_OBJECT *ob,
                           FL_MODIFY_CANVAS_PROP init,
                           FL_MODIFY_CANVAS_PROP activate,
                           FL_MODIFY_CANVAS_PROP cleanup);
```

where `init` will be called before the creation of the canvas window; `activate` is called once the window is created and `cleanup` is called before the window is destroyed. It is very convenient to set canvas attributes, such as depth and colormap etc (if different from the form), via the init routine.

This routine obviously should be called before the form is shown:

```
fd_form = create_form();
fl_modify_canvas_prop(fd_form->canvas, myInit, myActivate, myCleanup);
...
fl_show_form(fd_form->form, ...)
```

Given these services, creating a specialized canvas mainly consists of writing the three routines to be registered with the canvas handler. We start by writing the initialization routine

```
#include "forms.h"
#include <GL/glx.h>
#include <GL/gl.h>

static int config[] = {GLX_RGBA,GLX_DOUBLEBUFFER,GLX_DEPTH_SIZE,1,None};

int glx_init(FL_OBJECT *ob)
```

```
{
    XVisual *vi;
    GLXContext context;

    /* query for OpenGL capabilities */

    if(!glxQueryExtension(fl_display, 0, 0))
    {
       fprintf(stderr,"OpenGL is not supported\n");
       return -1;    /* signal the caller we have failed */
    }

    /* select the desired visual */

    if(!(vi = glxChooseVisual(fl_display, fl_screen, config)))
       return -1;

    /* change canvas visual/colormap based on what we've got */
    fl_set_canvas_visual(ob, vi->visual);
    fl_set_canvas_depth(ob, vi->depth);

    /* we need to create a colormap appropriate for the visual we get.
     * Also it is a good idea to fill it with xform's default
     * colors to reduce flashing in case the canvas visual is not
     * the same as the visual rest of the form is using
     */
    fl_set_canvas_colormap(ob, fl_create_colormap(vi, 1));

    if(!(context = glxCreateContext(fl_display, vi, None, GL_TRUE)))
    {
       fprintf(stderr,"Can't create GLX Context\n");
       return -1;
    }

    /* use the c_vdata field to store this context. Similar to
     * u_vdata, the main parts of the library does not reference or
     * modify c_vdata.
     */
    ob->c_vdata = context;
    return 0;
}
```

Routine `activate` and `cleanup` can be coded in a similar fashion

```
int glx_activate(FL_OBJECT * ob)
{
    glXMakeCurrent(fl_display, FL_ObjWin(ob), ob->c_vdata);
```

```
    return 0;
}

int glx_cleanup(FL_OBJECT * ob)
{
   if(ob->c_vdata)
       glXDestroyContext(fl_display, ob->c_vdata);
   ob->c_vdata = 0;
   return 0;
}
```

With the above routines in place, we write the glcanvas interface routine just like the interface routine for any other objects

```
FL_OBJECT *fl_create_glcanvas(int type, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h, const char *label)
{
   FL_OBJECT *ob = fl_create_canvas(type, x, y, w, h, label);
   fl_modify_canvas_prop(ob, glx_init, glx_activate, glx_cleanup);
   return ob;
}

FL_OBJECT *
fl_add_glcanvas(int type, FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
             const char *label)
{
   FL_OBJECT *ob = fl_create_glcanvas(type, x, y, w, h, label);
   fl_add_object(fl_current_form, ob);
   return ob;
}
```

Then the application program simply uses the glcanvas as an independent class.

Part IV

Designing your own object classes

Chapter 22

Introduction

Earlier chapters discussed ways to build user interfaces by combining suitable objects from the **Forms Library**, defining a few object callbacks and using Xlib functions. However, there is always a possibility that the built-in objects of the **Forms Library** might not be enough. Although free objects in principle provide all the flexibility a programmer needs, there are situations where it is beneficial to create new types of objects, for example, switches or joysticks or other types of sliders, etc. In this case, a programmer can use the architecture defined by the **Forms Library** to create the new object class that will work smoothly with the built-in or user-created object classes.

Creating such new object classes and adding them to the library is simpler than it sounds. In fact it is almost the same as making free objects. This part gives you all the details of how to add new classes. In chapter 23 a global architectural overview is given of how the **Forms Library** works and how it communicates with the different object classes by means of events (messages). Chapter 24 describes in detail what type of events objects can receive and how they should react to them. Chapter 25 describes in detail the structure of the type `FL_OBJECT` which plays a crucial role, a role equivalent to a superclass (thus all other object classes have `FL_OBJECT` as their parent class) in object-oriented programming.

One of the important aspects of an object is how to draw it on the screen. Chapter 26 gives all the details on drawing objects. The **Forms Library** contains a large number of routines that help you draw objects. In this chapter an overview is given of all of them. Chapter 27 gives an example illustrating how to create a new object class. Due to the importance of button classes, special routines are provided by the **Forms Library** to facilitate the creation of this particular class of objects. Chapter 28 illustrates by two examples the procedures of creating new button classes using the special services. One of the examples is taken from the **Forms Library** itself and the other offers actual usability.

Sometimes it might be desirable to alter the behavior of a built-in class slightly. Obviously a full-blown (re)implementation from scratch of the original object class is not warranted. Chapter 29.1 discusses the possibilities of using the pre-emptive handler of an object to implement derived objects.

Chapter 23

Global structure

The **Forms Library** defines the basic architecture of an object class. This architecture allows different object classes developed by different programmers to work together without complications.

The **Forms Library** consists of a main module and a number of object class modules. The object class modules are completely independent from the main module. So new object class modules can be added without any change (nor recompilation) of the main module. The main module takes care of all the global bookkeeping and the handling of events. The object class modules have to take care of all the object specific aspects, like drawing the object, reacting to particular types of user actions, etc. For each class there exists a file that contains the object class module. For example, there are files `slider.c, box.c, text.c, button.c`, etc.

The main module communicates with the object class modules by means of events (messages if you prefer). Each object has to have a handle routine known to the main module so that it can be called whenever something needs to be done. One of the arguments passed to the handle routine is the type of event, e.g. `FL_DRAW`, indicating that the object needs to be redrawn.

Each object class consists of two components. One component, both its data and functions, is common to *all* object classes in the **Forms Library**. The other component is specific to the object class in question and is typically opaque. So for typical object classes, there should be routines provided by the object class to manipulate the object class specific data. Since C lacks inheritance as a language construct, inheritance is implemented in the **Forms Library** by pointers and the global function `fl_make_object()`.[1] It is helpful to understand the global architecture and the object-oriented approach of the **Forms Library**, it makes reading the C code easier and also adds perspective on why some of the things are implemented the way they are.

In this chapter it is assumed that we want to create a new class with a name `NEW`. Creating a new object class mainly consists of writing the handle routine. There also should be a routine that adds an object of the new class to a form and associates the handle routine to it. This routine should have the following basic form:

```
FL_OBJECT *fl_add_NEW(int type,FL_Coord x,FL_Coord y,
                      FL_Coord w,FL_Coord h, const char *label)
```

[1]There are other ways to simulate inheritance, such as including a pointer to generic objects as part of the instance specific data

This routine must add an object of class NEW to the current form. It gets the parameters `type`, indicating the type of the object within the class (see below), `x`, `y`, `w`, and `h`, indicating the bounding box of the object in the current active unit (mm, point or pixels), and `label` which is the label of the object. This is the routine the programmer uses to add an object of class NEW to a form. See below for the precise actions this routine should take.

One of the tasks of `fl_add_NEW()` is to bind the event handling routine to the object. For this it will need a routine:

```
static int handle_NEW(FL_OBJECT *obj,int event,FL_Coord mx,FL_Coord my,
                         int key, void *xev)
```

This routine is the same as the handle routine for free objects and should handle particular events for the object. `mx`, `my` give the current mouse position and *key* the key that was pressed (if this information is related to the event). See chapter 24 for the types of events and the actions that should be taken. `xev` is the XEvent that caused the invocation of the handler. Note that some of the events may have a null `xev` parameter, so `xev` should be checked before dereferencing it.

The routine should return whether the status of the object is changed, i.e., whether the event dispatcher should invoke this object's callback or if no callback whether the object is to be returned to the application program by `fl_do_forms()` or `fl_check_forms()`. What constitutes a status change is obviously dependent on the specific object class and possibly its types within this class. For example, a mouse push on a radio button is considered a status change while it is not for a normal button where a status change occurs on release.

Moreover, most classes have a number of other routines to change settings of the object or get information about it. In particular the following two routines often exist:

```
void fl_set_NEW(FL_OBJECT *obj, ...)
```

that sets particular values for the object and

```
??? fl_get_NEW(FL_OBJECT *obj, ...)
```

that returns some particular information about the object. See e.g. the routines `fl_set_button()` and `fl_get_button()`.

23.1 The routine `fl_add_NEW()`

`fl_add_NEW()` has to add a new object to the form and bind its handle routine to it. To make it consistent with other object classes and also more flexible, there should in fact be two routines: `fl_create_NEW()` that creates the object and `fl_add_NEW()` that actually adds it to the form. They normally look as follows:

```
typedef struct { /* instance specific record */} SPEC;
```

```
FL_OBJECT *fl_create_NEW(int type,FL_Coord x,FL_Coord y,
                         FL_Coord w,FL_Coord h,const char *label)
{
    FL_OBJECT *ob;

    /* create a generic object */
    ob = fl_make_object(FL_COLBOX,type,x,y,w,h,label,handle_NEW);

    /* fill in defaults */
    ob->boxtype = FL_UP_BOX;

    /* allocate instance-specific storage and fill it with defaults */
    ob->spec_size = sizeof(SPEC);
    ob->spec = fl_calloc(1, op->spec_size);
    return ob;
}
```

The constant `FL_NEW` will indicate the object class. It should be an integer. The numbers 0–(`FL_USER_CLASS_START-1`)(1000) and `FL_BEGIN_GROUP`(10000) and higher are reserved for the system and should not be used. Also it is preferable to use `fl_malloc()`, `fl_calloc()`, `fl_realloc()` and `fl_free()` to allocate/free the memory for the instance specific structures. These routines have the same prototypes and work the same as those in the standard library, and may offer additional debugging capabilities in future versions of **Forms Library**.

The pointer `ob` returned by `fl_make_object()` will have all of its fields set to some defaults (See Chapter 25). In other words, the newly created object inherits many attributes of a generic one. Any class specific defaults that are different from the generic one can be changed after `fl_make_object()`. Conversion of unit, if different from the default `pixel`, is performed within `fl_make_object()` and a class module never needs to know what the prevailing unit is. After the object is created, it has to be added to a form:

```
FL_OBJECT *fl_add_NEW(int type,FL_Coord x,FL_Coord y,FL_Coord w,
                      FL_Coord h, const char *label)
{
    FL_OBJECT *ob;
    ob = fl_create_NEW(type,x,y,w,h,label);
    fl_add_object(fl_current_form,ob);
    return ob;
}
```

Chapter 24

Events

As indicated above, the main module of the **Forms Library** communicates with the objects by calling the associated handling routine with, as one of the arguments, the particular event for which action must be taken. In the following we assume that `obj` is the object to which the event is sent. The following types of events can be sent to an object:

`FL_DRAW` The object has to be (re)drawn. To figure out the size of the object you can use the fields `obj->x`, `obj->y`, `obj->w` and `obj->h`. Many Xlib drawing routines require a window ID, which you can obtain from the object pointer using `FL_ObjWin(ob)`. Some other aspects might also influence the way the object has to be drawn. E.g., you might want to draw the object differently when the mouse is on top of it or when the mouse is pressed on it. This can be figured out as follows. The field `obj->belowmouse` indicates whether the object is below the mouse. The field `obj->pushed` indicates whether the object is currently being pushed with the mouse. Finally, `obj->focus` indicate whether input focus is directed towards this object. Note that the drawing of the object is the full responsibility of the object class, including the bounding box and the label, which can be found in the field `obj->label`. The **Forms Library** provides a large number of routines to help you draw object. See chapter 26 for more details on drawing objects and an overview of all available routines.

One caution about the draw event handle code is that all the high level routines (`fl_freeze_form()`, `fl_deactivate_form()`) should not be used. The only routines allowed are (direct) drawing and object internal book keeping routines. Attributes modifying routines, such as `fl_set_object_color()` etc. are not allowed (which can lead to infinite recursion). In addition, (re)drawing of other objects using `fl_redraw_object()` while handling `FL_DRAW` would not work.

Due to the way the double buffering is handled, `FL_ObjWin(ob)` at `FL_DRAW` time (and only then) is the backbuffer if the object is double bufferred. What this means is that `FL_ObjWin(ob)` should not be used where a real window is expected. The difference between xforms backbuffer and a real window is that you can change the real window's cursor or query the mouse position with it. You can not do either of these with the backbuffer pixmap. If there is a need to obtain the real window ID, the following routine can be used:

```
Window fl_get_real_object_window(FL_OBJECT *)
```

To summarize, use `FL_ObjWin(ob)` when drawing and use `fl_get_real_object_window()` for cursor or pointer routines. This distinction is important only while handling `FL_DRAW` and `FL_ObjWin(ob)` should be used anywhere else.

`FL_DRAWLABEL` This event typically follows `FL_DRAW` and indicates the object label needs to be (re)drawn. If the object in question always draws its label inside the bounding box, and is taken care of by `FL_DRAW`, you can ignore this event.

`FL_ENTER` This event is sent when the mouse has entered the bounding box. This might require some action. Note also that the field `belowmouse` in the object is being set. If entering only changes the appearance, redrawing the object normally suffices. **Don't** do this directly! Always redraw the object using the routine `fl_redraw_object()`. It will send an `FL_DRAW` event to the object but also does some other things (like setting window id's and taking care of double buffering).

`FL_LEAVE` The mouse has left the bounding box. Again, normally a redraw is enough (or nothing at all).

`FL_MOTION` A motion event is sent between `FL_ENTER` and `FL_LEAVE` events when the mouse position changes on the object (in fact, it is sent all the time even if the mouse position remains the same). The mouse position is given as an argument to the handle routine.

`FL_PUSH` The user has pushed a mouse button in the object. Normally this requires some actual action. The number of the mouse button pushed is given in the `key` parameter. 1 = leftmouse, 2 = middlemouse, 3 = rightmouse.

`FL_RELEASE` The user has released the mouse button. This event is only sent if a PUSH event was sent earlier. The number of the mouse button released is given in the `key` parameter. 1 = leftmouse, 2 = middelmouse, 3 = rightmouse.

`FL_DBLCLICK` The user has pushed a mouse button twice within a certain time limit (`FL_CLICK_TIMEOUT`). This event is sent after two `FL_PUSH`, `FL_RELEASE` sequence. Note that `FL_DBLCLICK` is only generated for objects that have non-zero `obj->click_timeout` fields and it will not be generated for middle mouse button clicks.

`FL_TRPLCLICK` The user has pushed a mouse button three times within a certain time window between each push. This event is sent after a `FL_DBLCLICK`, `FL_PUSH`, `FL_RELEASE` sequence. Set `click_timeout` to none zero to activate `FL_TRPLCLICK`.

`FL_MOUSE` This event is sent to an object between an `FL_PUSH` and an `FL_RELEASE` event (i.e., a mouse button is down). The mouse position is given with the routine and action can be taken. For example, sliders use this event while buttons do not. Note that this event is send periodically as long as the a mouse button is down.

`FL_FOCUS` Input got focussed to this object. This type of event and the next two are only sent to an object for which the field `obj->input` is set to 1 (see below).

`FL_UNFOCUS` Input is no longer focussed on this object.

FL_KEYBOARD A key was pressed. The ASCII value (or KeySym if non-ASCII) is given with the routine. This event only happens between FL_FOCUS and FL_UNFOCUS events. Not all objects are sent keyboard events, only those that have non-zero value in field obj->input or obj->wantkey.

FL_STEP A step event is sent all the time (typically 20 times a second but often less because of system delays and other time-consuming tasks, e.g. a time-consuming redraw) to an object if the field obj->automatic has been set to 1. This can be used to make an object change appearance without user action. E.g. the clock uses these type of events.

FL_SHORTCUT The user used a keyboard shortcut. The shortcut used is given in the parameter key. See below for more on shortcuts.

FL_FREEMEM This event is sent when the object is to be freed. All memory allocated by the object class should be freed when this event is received.

FL_OTHER Events other than the above. These events currently include ClientMessage, Selection and possibly other window manager events. All information about the event is contained in xev parameter and mx,my may or may not reflect the actual position of the mouse.

Many of these events might make it necessary that the object has to be redrawn or partially redrawn. Always do this using the routine fl_redraw_object().

24.1 Shortcuts

The **Forms Library** has a mechanism of dealing with keyboard shortcuts. In this way the user can use the keyboard rather than the mouse for particular actions. Obviously only active objects can have shortcuts. At the moment there are three object classes that use this, namely buttons, inputs and browsers although they behave differently.

The mechanism works as follows. There is a routine

```
void fl_set_object_shortcut(FL_OBJECT *obj, const char *str, int showit)
```

with which the object class can bind a series of keys to an object. E.g., when str is "acE#d^h" the keys a,c,E, <ALT> d and <CNTRL> h are associated with the object. The precise format is as follows: Any character in the string is considered as a shortcut, except for ^ and #, which stand for combinations with the <CONTROL>, and <ALT> key. (There is no difference between e.g. ^C and ^c.) The symbol ^ itself can be obtained using ^^. The symbol # can be obtained using ^#. So, e.g. #^# means <ALT> #. The <ESCAPE> key can be given as ^[.

To indicate function and arrow keys, the &n sequence (n = 1 . . . 35) can be used. For example, &2 indicates <F2> key. Note that the four cursors keys (up, down, right, and left) can be given as <&A>, <&B>, <&C> and <&D> respectively. The key & itself can be obtained by prefixing it with ^.

Parameter `showit` indicates whether the shortcut letter in the object label should be underlined if a match exists. Although the entire object label is searched for matches, only the first alphanumerical character in the shortcut string is used. E.g., for object label `"foobar"`, shortcut `"oO"` would result in a match at the first `o` in `"foobar"` while `"Oo"` would not. However, `"^O"` always matches.

To use other special keys not described above as shortcuts, the following routine must be used

```
void fl_set_object_shortcutkey(FL_OBJECT *ob, unsigned int key)
```

where <KEY> is an X KeySym, for example, `XK_Home`, `XK_F1` etc. Note that function `fl_set_object_shortcutkey` always appends the key specified to the current shortcuts while `fl_set_object_shortcuts` resets the shortcuts. Of course, special keys can't be underlined.

Now whenever the user presses one of these keys an `FL_SHORTCUT` event is sent to the object. Here the key pressed is given with the handle routine (in the argument `key`). Combinations with the <ALT> key are given by adding `FL_ALT_VAL` (currently the 25th bit, i.e., 0x1000000) to the ASCII value of the rest. E.g. `#^E` is passed as 5+`FL_ALT_VAL`. The object can now take action accordingly. If you use shortcuts to manipulate class object specific things, you will need to create a routine to communicate with the user, e.g., `fl_set_NEW_shortcut()`, and do your own internal bookkeeping to track what keys do what and then call `fl_set_object_shortcut()` to register the shortcut in the event dispatching module. The idea is NOT that the user himself calls `fl_set_object_shortcut()` but that the class provides a routine for this that also keeps track of the required internal bookkeeping. Of course, if there is no internal bookkeeping, a macro to this effect would suffice. For example, `fl_set_button_shortcut` is defined as `fl_set_object_shortcut`.

The order in which keys are handled is as follows: First a key is tested whether any object in the form has the key as a shortcut. If affirmative, the first of those objects gets the shortcut event. Otherwise, the key is checked to see if it is <TAB> or <RETURN>. If it is, the `obj->wantkey` field is checked. If the field does not contain `FL_KEY_TAB` bit, input is focussed on the next input field. Otherwise the key is sent to the current input field. This means that input objects only get a <TAB> or <RETURN> key sent to them if the field `obj->wantkey` contain `FL_KEY_TAB`. This is e.g. used in multi-line input fields. If the object wants all cursor keys (including <PGUP> etc.), the `wantkey` field can be set to `FL_KEY_SPECIAL`.

To summarize, the `obj->wantkey` can take on the following values or the bit-wise `or` of them

FL_KEY_NORMAL The default. Left and right cursor keys, <HOME> and keyEnd plus all normal keys (0-255) except for <TAB> and keyReturn.

FL_KEY_TAB `FL\_KEY\_NORMAL` plus <TAB>, <RETURN> and Up and Down cursor keys.

FL_KEY_SPECIAL All special keys (> 255).

FL_KEY_ALL All keys.

It is possible for a non-input object (i.e.,`obj->input` is zero) to obtain special keyboard event by setting `obj->wantkey` to `FL_KEY_SPECIAL`.

Chapter 25

The type `FL_OBJECT`

Each object has a number of attributes. Some of them are used by the main routine, some have a fixed meaning and should never be altered by the class routines and some are free for the class routines to use. Below we consider some of them that are likely to be used in new classes.

`objclass` This indicates the class of the object (E.g.,`FL_BUTTON`, `FL_NEW` etc.)

`type` This indicates the type of the object within the class. Types are integer constants that should be defined in the file NEW.h. Their use is completely free. For example, in the class slider the type is used to distinguish between horizontal and vertical sliders. At least one type should exist and the user should always provide it (just for consistency). They should be numbered from 0 upwards.

`boxtype` This is the type of the bounding box for the object. The routine `handle_NEW` has to take care that this is actually drawn. Note that there is a routine for this, see below.

`x,y,w,h` These are Coord's that indicate the bounding box of the object. They always have to be provided when adding an object. The system uses them to determine the object below the mouse. The class routines should use them to draw the object in the correct size, etc. Note that these values will change when the user resizes the form window. So never assume anything about their values but always recheck them when drawing the object.

`resize` An integer controlling if the object should be resized if the form it is on is resized. The options are `FL_RESIZE_NONE`, `FL_RESIZE_X` and `FL_RESIZE_Y`. Default is `FL_RESIZE_X|FL_RESIZE_Y`.

`nwgravity,segravity` These two variables control how the object should be placed relative to its position prior to resizing.

`col1,col2` These are two color indices in the internal color lookup table. The class routines are free to use them or not. The user can provide them using the routine `fl_set_object_color()`. The routine `fl_add_NEW()` should fill in defaults.

`label` This is a pointer to a text string. This can be used by class routines to provide a label for the object. The class routines can freely use this. (Don't forget allocating storage for it when you want to set it yourself, i.e., when you don't use `fl_set_object_label()`.) The user can change it using the routine `fl_set_object_label()`. The label must be drawn by the routine `handle_NEW` when it receives a `FL_DRAW` event. (The system does not draw the label automatically because it does not know where to draw it.) For non-offsetted labels, i.e., the alignment is relative to the entire bounding box, simply calling `fl_draw_object_label()` should be enough.

`lcol` The color of the label. The class routines can freely use this. The user sets it with `fl_set_object_lcol()`.

`lsize` The size of the label. The class routines can freely use this. The user sets it with `fl_set_object_lsize()`.

`lstyle` The style of the label, i.e. the number of the font in which it should be drawn. The class routines can freely use this. The user sets it with `fl_set_object_lstyle()`.

`align` The alignment of the label with respect to the object. Again it is up to the class routines to do something useful with this. The possible values are `FL_ALIGN_LEFT`, `FL_ALIGN_RIGHT`,
`FL_ALIGN_TOP`, `FL_ALIGN_BOTTOM`, `FL_ALIGN_CENTER`. `FL_ALIGN_TOP_LEFT`, `FL_ALIGN_TOP_RIGHT`, `FL_ALIGN_BOTTOM_LEFT` and `FL_ALIGN_BOTTOM_RIGHT`. The user can set this using the routine `fl_set_object_align()`.

`bw` An integer indicating the border width of the object. Negative indicates the up box should look "softer"

`shortcut` A pointer to long containing all shortcuts (as keysyms) defined for the object. (See the previous section.) You should never need them because they are fully handled by the main routines.

`spec` This is a pointer that points to any class specific information. The `fl_add_NEW()` routine will have to provide storage for it. For example, for sliders it stores the minimum value, maximum value and current value of the slider. Most classes (except the most simple ones like boxes and texts) will need this. Whenever the object receives the event `FL_FREEMEM` it should free this memory.

`visible` Indicates whether the object is visible. The class routines don't have to do anything with this variable. When the object is not visible the main routine will never try to draw it or send events to it. By default objects are visible. Note that a true `visible` does not guarantee the object is visible on the screen, for that the form need to be also visible, i.e., `fl_form_is_visible()` is true.

`active` Indicates whether the object is active, i.e., wants to receive events other than `FL_DRAW`. Static objects, such as text and boxes are inactive. Changing the status should be done in the `fl_add_NEW()` routine if required. By default objects are active.

`input` Indicates whether this object can receive keyboard input. If not, events that are related to keyboard input are not sent to the object. The default `input` is false. It should be set by `fl_add_NEW()` if required. Note that not all keys are sent (see `wantkey` below).

`wantkey` An input object normally does not receive <TAB> or <RETURN> keystrokes or any other keys except those that have values between 0-255 and left- and right-arrows (<TAB> and <RETURN> are reserved and used to switch between input objects). By setting this field to `FL_KEY_TAB` these keystrokes as well as as four directional cursor keys will also be sent to the object when focus is directed to it. If however, an object is only interested in keys that are special (e.g., <HOME>,<PGUP> etc), this variable can be set to `FL_KEY_SPECIAL` with or without `input` being set.

`click_timeout` If non-zero, it indicates the the maximum elapsed time between two mouse clicks to be considered a double click. A zero value disables double/triple click detection.

`radio` This indicates whether this object is a radio object. This means that, whenever it is pushed, other radio objects in the same group in the form that are pushed are released (and their pushed value is reset). Radio buttons use this. The default is false. The `fl_add_NEW()` routine should set it if required.

`automatic` An object is automatic if it automatically (without user actions) has to change its contents. Automatic objects get a `FL_STEP` event all the time. For example, the object class clock is automatic. `automatic` by default is false.

`belowmouse` This indicates whether the mouse is on this object. It is set and reset by the main routine. The class routines should never change it but can use it to draw or handle the object differently.

`pushed` This indicates whether the mouse is pushed within the bounding box of the object. It is set and reset by the main routine. Class routines should never change it but can use it to draw or handle objects differently.

`focus` Indicates whether keyboard input is sent to this object. It is set and reset by the main routine. Never change it but you can use its value.

`handle` This is a pointer to the interaction handling routine. `fl_add_NEW()` sets this by providing the correct handling routine. Normally it is never used or changed although there might be situations in which you want to change the interaction handling routine for an object, due to some user action.

`next,prev,form` These are pointers to other objects in the form and to the form itself. They are used by the main routines. The class routines should not change them.

`c_vdata` A void pointer for the class routine. The main module does not reference or modify this field in any way. The object classes, including the built-in ones, may use this field.

`c_cdata` A char pointer for the class routine. The main module does not reference or modify this field in any way. The object classes, including the built-in ones, may use this field.

`c_ldata` A long variable for the class routine. The main module does not reference or modify this field in any way. The object classes, including the built-in ones, may use this field.

`u_vdata` A void pointer for the application program. The main module does not reference or modify this field in any way and neither should the class routines.

u_cdata A char pointer for the application program. The main module does not reference or modify this field in any way and neither should the class routines.

u_ldata A long variable provided for the application program.

object_callback The call-back routine that the application program assigns to the object. This is the responsibility of the application program and the class routines should not use it.

argument The argument to the call-back routine. Again, this is the responsibility of the application program to set.

The generic object construction routine

```
FL_OBJECT *fl_make_object(int objclass, int type,
                          FL_Coord x, FL_Coord y, FL_Coord w, FL_Cord h,
                          const char *label, FL_HANDLEPTR handle)
```

allocates a chunk of memory appropriate for all object classes and initializes the newly allocated object to the following state:

```
obj->resize = FL_RESIZE_X|FL_RESIZE_Y;
obj->nwgravity = obj->segravity = FL_NoGravity;
obj->boxtype = FL_NO_BOX;
obj->align = FL_ALIGN_CENTER | FL_ALIGN_INSIDE;
obj->lcol  = FL_BLACK;
obj->lsize = FL_DEFAULT_SIZE;   /* SMALL_SIZE, 10pt */
obj->lstyle = FL_NORMAL_STYLE;
obj->col1 = FL_COL1;
obj->col2 = FL_MCOL;
obj->wantkey == FL_KEY_NORMAL;
obj->active = 1;
obj->visible = 1;
obj->bw = (borderWidth resource set ? resource_val:FL_BOUND_WIDTH);
obj->u_ldata = 0;
obj->u_vdata = 0;
obj->spec = 0;
```

There is rarely any need for the new object class to know how the object is added to a form and how the **Forms Library** manages the geometry, e.g., does an object have its own window etc. Nonetheless if this information is required, use FL_ObjWin(ob) to obtain the window resource ID the object belongs to. Beware that an object window ID may be shared with other objects[1]. Always remove an object from the screen with fl_hide_object().

The class routine/application may reference the following members of the FL_FORM structure to obtain information on the status of the form, but should not modify them directly

[1]the only exception is the canvas class where the window ID is guaranteed to be non-shared

`int visible` indicates if the form is visible on the screen (mapped). Use `fl_show_form()` and/or `fl_hide_form()` to change this member.

`int deactivated` indicates if the form is deactivated.

`FL_OBJECT *focusobj` This pointer points to the object on the form that has the input focus.

`FL_OBJECT *first` The first object on the form. Pointer to a linked list.

`Window window` The form window.

Chapter 26

Drawing objects

An important aspect of a new object class (or a free object) is how to draw it. As indicated above this should happen when the event `FL_DRAW` is received by the object. The place, i.e. bounding box, where the object has to be drawn is indicated by the fields `obj->x`, `obj->y`, `obj->w` `obj->h`. Forms are drawn in the **Forms Library** default visual or the user requested visual, which could be any of the X supported visuals. Hence, preferably your classes should run well in all visuals. **Forms Library** tries to hide as much as possible the information about graphics mode, and in general, using the built-in drawing routines is the best approach. Here are some details about graphics state in case such information is needed.

All state information is kept in a global structure of type `FL_STATE` and there is a total of six (6) such structures `fl_state[6]`, each for every visual class. The structure contains the following members, among others

`XVisualInfo *xvinfo` Many properties of the current visual can be obtained from this member.

`int depth` The depth of the visual. Same as what you get from `xvinfo`.

`int vclass` The visual class, `PseudoColor, TrueColor` etc.

`Colormap colormap` Current active colormap valid for the current visual for the entire **Forms Library** (except `FL_CANVAS`). You can allocate colors from this colormap, but you should *never* free it.

`Window trailblazer` This is a valid window resource ID created in the current visual with the colormap mentioned above. This member is useful if you have to call, before the form becomes active (thus don't have a window ID), some Xlib routines that require a valid window. A macro, `fl_default_window()`, is defined to return this member and use of the macro is encouraged.

`GC gc[16`] total of 16 GCs appropriate for the current visual and depth. The first (`gc[0]`) is the default GC used by many internal routines and should be modified with care. It is a good idea to use only the top 8 GCs (8-15) for your free object so that future **Forms Library** extensions won't interfere with your program. Since many internal drawing routines use the **Forms Library**'s default GC (gc[0]), it can change anytime whenever drawing occurs. Therefore, if you are using this GC for some of your own drawing routines make sure to always set the proper value before using it.

Currently active visual class `TrueColor, PseudoColor` etc. can be obtained by the following function/macro:

```
int fl_get_form_vclass(FL_FORM *);

int fl_get_vclass(void);
```

The value returned can be used as an index into the `fl_state` structure. Note `fl_get_vclass()` should only be used within a class/new object module where there can be no confusion what the "current" form is.

Other information about the graphics mode can be obtained by using visual class as an index into the `fl_state` structure. For example, to print the current visual depth, code similar to the following can be used:

```
int vmode = fl_get_vclass();
printf("depth: %d\n", fl_state[vmode].depth);
```

Note that `fl_state[]` for indices other than the currently active visual class might not be valid.

In almost all Xlib calls, the connection to the X server and current window ID are needed. **Forms Library** maintains some utility functions/macros to facilitate easy utilization of Xlib calls. Since the current version of **Forms Library** only maintains a single connection, the global variable `Display *fl_display` can be used where required. However, it is recommended that you use `fl_get_display()` or `FL_FormDisplay(form)` instead since the function/macro version has the advantage that your program will remain compatible with future (possibly multi-connection) versions of the **Forms Library**.

There are a couple of ways to find out the "current" window ID, defined as the window ID the object receiving dispatcher's messages `FL_DRAW` etc. belongs to. If the object ID is available, `FL_ObjWin(obj)` would suffice and otherwise, `fl_winget()` can be used.

There are other routines that might be useful:

```
FL_FORM *fl_win_to_form(Window win)
```

This function takes a window ID `win` and returns the form the window belongs to either as an equivalent `form->window == win` or as a child to `form->window`.

As mentioned earlier, **Forms Library** keeps an internal colormap initialized to predefined colors. The predefined color symbols do not correspond to pixel values the server understands. Therefore, they should never be used in any of the GC altering or Xlib routines. To get the actual pixel value the server understands, use the following routine

```
FL_COLOR fl_get_pixel(FL_COLOR index)
```

e.g., to get the pixel value of red color, use

```
FL_COLOR red_pixel;
red_pixel = fl_get_pixel(FL_RED);
```

Or more conveniently

```
fl_color(FL_RED);
```

This sets the foreground color in the default GC (`gc[0]`) to `red_pixel`.

To set the background color in the **Forms Library**'s default GC, use the follow routine

```
fl_bk_color(FL_COLOR index)
```

To set foreground or background in GCs other than the **Forms Library**'s default, the following functions exist:

```
void fl_set_foreground(GC gc, FL_COLOR index)
void fl_set_background(GC gc, FL_COLOR index)
```

which is equivalent to the following Xlib calls

```
XSetForeground(fl_display, gc, fl_get_pixel(index))
XSetBackground(fl_display, gc, fl_get_pixel(index))
```

To free allocated colors from the default colormap, use the following routine

```
void fl_free_colors(FL_COLOR *cols, int n);
```

This function frees the colors represented by the `cols` array.

In case the pixel values, as opposed to **Forms Library**'s values, are known, the following routine can be used to free the colors from the default colormap

```
void fl_free_pixels(unsigned long *pixels, int n);
```

Note that the internal colormap maintained by the **Forms Library** is not updated. This is in general harmless.

To modify or query the internal colormap, use the following routines,

```
long fl_mapcolor(FL_COLOR ind, int red, int green, int blue)
long fl_mapcolorname(FL_COLOR ind, const char *name)

void fl_getmcolor(FL_COLOR ind, int *red, int *green, int *blue)
```

The coordinate system of the form by default corresponds directly to the screen. Hence a pixel on the screen always has size 1 in the default coordinate system of the form. Object coordinates are relative to the upper-right corner of the form.

To obtain the position of the mouse in the current form/window, use the routine

```
Window fl_get_form_mouse(FL_FORM *form, FL_Coord *x, FL_Coord *y,
                         unsigned *keymask)

Window fl_get_win_mouse(Window win, FL_Coord *x, FL_Coord *y,
                        unsigned *keymask)
```

The functions return the window ID the mouse is in. Upon its return, `x,y` would be set to the the mouse position relative to the form/window, and `keymask` contains information on modifier keys (same as the the corresponding `XQueryPointer()` argument).

Similar routine exists that can be used to obtain the mouse location relative to the root window

```
Window fl_get_mouse(FL_Coord *x, FL_Coord *y, unsigned *keymask);
```

The function returns the window ID the mouse is in.

To move the mouse to a specific location relative to the root window, use the following routine

```
void fl_set_mouse(FL_Coord x, FL_Coord y)
```

To avoid drawing outside a bounding box the following routine exists.

```
void fl_set_clipping(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h)
```

It sets a clipping region in the **Forms Library**'s default GC. `x`, `y`, `w` and `h` are as in the definition of objects. Drawing is restricted to this region after the call. In this way you can prevent drawings from sticking into other objects. Always use after drawing

```
void fl_unset_clipping(void)
```

to stop clipping.

To obtain the bounding box of an object with the dimension and location of the label taken into account (compare with `fl_get_object_geometry()` the following routine exists:

```
void fl_get_object_bbox(FL_OBJECT *ob, FL_Coord *x, FL_Coord *y,
                        FL_Coord *w, FL_Coord *h)
```

To set clippings for text, which uses a different GC, the following routine should be used

```
void fl_set_text_clipping(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h)

void fl_unset_text_clipping(void)
```

For drawing text at the correct places you will need some information about the sizes of characters and strings. The following routines are provided:

```
int fl_get_char_height(int style, int size, int *ascend, int *descend)


int fl_get_char_width(int style, int size)
```

These two routines return the maximum height and width of the font used, where `size` indicates the point size for the font and `style` is the style in which the text is to be drawn. A list of valid styles can be found in Section 3.11.3. To obtain the width and height information on a specific string, use the following routines

```
int fl_get_string_width(int style, int size, const char *str, int len)

int fl_get_string_height(int style, int size, const char *str, int len,
                         int *ascend, int *descend)
```

where `len` is the string length. The functions return the width and height of the string `str` respectively.

There exists also a routine that returns the width and height of a string in one call. In addition, the string passed can contain embedded newline in it and the routine will make proper adjustment so the values returned are (just) large enough to contain the multiple lines of text

```
void fl_get_string_dimension(int style, int size, const char *str,
                             int len, int *width, int *height)
```

Sometimes, it may be useful to get the X font structure for a particular size and style as used in **XForms**. For this purpose, the following routine exists:

```
[const] XFontStruct *fl_get_fontstruct(int style, int size)
```

The structure returned can be used in, say, setting the font in a particular GC

```
XFontStruct *xfs = fl_get_fontstruct(FL_TIMESBOLD_STYLE, FL_HUGE_SIZE);
   XSetFont(fl_get_display(), mygc, xfs->fid);
```

Caller should not free the structure returned by `fl_get_fontstruct()`.

There are a number of routines that help you draw objects on the screen. All **XForms**'s internal drawing routine draws into the "current window", defined as the window the object that uses the drawing routine belongs to. Nevertheless, the following routines can be used to set or query the current window

```
void fl_winset(Window win)

Window fl_winget(void)
```

One caveat about `fl_winget()` is that it can return 0 if called outside of object's event handler depending on where the mouse is. Thus, the return value of this function should be checked when called outside of an object handler.

It is important to remember that unless the following drawing commands are issued while handling the `FL_DRAW` event (not generally recommended), it is the application's responsibility to set the proper drawable using `fl_winset()`.

The most basic drawing routine is the rectangle routines:

```
void fl_rectf(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)
void fl_rect(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)
```

Both draw a rectangle on the screen in color `col`. The difference is that `fl_rectf()` draws a filled rectangle while `fl_rect()` draws an outline.

To draw a filled rectangle with a black border, use the following routine

```
void fl_rectbound(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)
```

To draw a rectangle with rounded corners, the following routines exist

```
void fl_roundrectf(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                   FL_COLOR col)

void fl_roundrect(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                  FL_COLOR col)
```

To draw a general polygon, use one of the following routines

```
typedef struct {short x,y;} FL_POINT

void fl_polyf(FL_POINT *xpoint, int n, FL_COLOR col);

void fl_polyl(FL_POINT *xpoint, int n, FL_COLOR col);

void fl_polybound(FL_POINT *xpoint, int n, FL_COLOR col);
```

`fl_polyf()` draws a filled polygon; `fl_polyl()` draws a polyline; and `fl_polybound()` draws a filled polygon with a black outline. *Note all polygon routines require that* `xpoint` *have spaces to hold* `n+1` *points*.

To draw an ellipse, either filled or open, the following routines can be used (use `w == h` to get a circle)

```
void fl_ovalf(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)

void fl_ovall(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)

void fl_ovalbound(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                  FL_COLOR c)
```

To draw circular arcs, either open or filled, the following routines can be used

```
void fl_arc(FL_Coord x, FL_Coord y, FL_Coord radius,
            int start_theta, int end_theta, FL_COLOR col)

void fl_arcf(FL_Coord x, FL_Coord y, FL_Coord radius,
            int thetai, int thetaf, FL_COLOR col)
```

where `thetai` and `thetaf` are the starting and ending angles of the arc, in unit of one tenth of a degree (1/10 degree); and `x,y` are the center of the arc. If $\theta_f - \theta_i$ is larger than 3600 (360 degrees), it is truncated to 360 degrees.

To draw elliptical arcs, the following routine should be used

```
void fl_pieslice(int fill, FL_Coord x, FL_Coord y, FL_Coord w,
                 FL_Coord h, int theta1, int thetaf, FL_COLOR col)
```

The center of the arc is the center of the bounding box specified by (`x`, `y`, `w`, `h`) and `w` and `h` specify the major and minor axes respectively. `theta1` and `theta2`, measured in one tenth of a degree, specify the starting and ending angles measured from zero degrees (3 o'clock).

Depending on circumstance, elliptical arc may be more easily drawn using the following routine

```
void fl_ovalarc(int fill, FL_Coord x, FL_Coord y, FL_Coord w,
                 FL_Coord h, int theta, int dtheta, FL_COLOR col)
```

Here `theta` specifies the starting angle, again, measured in one tenth of a degree, relative to 3 o'clock position and `dtheta` specifies *both* the direction and extent of the arc. If `dtheta` is positive, it indicates counter-clockwise motion otherwise clockwise. The magnitude of `dtheta` is greater than 3600, it is truncated to 3600.

To connect two points with a straight line, use the following routine

```
void fl_line(FL_Coord x1, FL_Coord y1,
             FL_Coord x2, FL_Coord y2, FL_COLOR col)
```

There is also a routine to draw a line along the diagonal of a box (to draw a horizontal line use `h` = 1 not 0.)

```
void fl_diagline(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                 FL_COLOR col)
```

To draw multiple connected line segments, use the following routine

```
typedef struct {short x, y} FL_POINT;

void fl_lines(FL_POINT *points, int npoint, FL_COLOR col)
```

All coordinates in `points` are relative to the origin of the drawable.

There are also routines to draw pixel or pixels

```
void fl_point(FL_Coord x, FL_Coord y, FL_COLOR col)

void fl_points(FL_POINT *p, int np, FL_COLOR col)
```

Where all coordinates are relative to the origin of the drawable. Note that these routines are meant for you to draw a few pixels, not images consisting of tens of thousands of pixels of varying colors, for that `XPutImage()` (See *XPutImage*(3X11)) should be used. Also whenever possible when drawing multiple points, use `fl_points()` even if that means the application program has to pre-sort and group the like colored pixels first.

To change line width or style, the following convenience functions are available

```
void fl_linewidth(int lw)
void fl_linestyle(int style)
```

Use `lw=0` to reset line width to the server default. Line styles can take on the following values (see *XChangeGC*(3X11))

`FL_SOLID` Solid line. The most efficient.

`FL_DOT` Dotted line.

`FL_DASH` Dashed line

`FL_DOTDASH` Dash-dot-dash line.

`FL_LONGDASH` Long dashed line.

`FL_USERDASH` Dashed line, but the dash pattern is user definable via `fl_dashedlinestyle()`. Only the odd numbered segments are drawn with the foreground color.

`FL_USERDOUBLEDASH` Similar to `FL_LINE_USERDASH`, but both even and odd numbered segments are drawn with the even numbered segments drawn in background color (`fl_bk_color()`).

The following routine can be used to change the dash patterns of `FL_LINE_USERDASH` drawing request:

```
void fl_dashedlinestyle(const char *dash, int ndashes)
```

The meanings of the parameters are as follows: Each element of `dash` is the length of a segment of the pattern in pixels. Dashed lines are drawn as alternating segments, each of an element in `dash`. Thus the overall length of the dash pattern, in pixels, is the sum of all elements in `dash`. When the pattern is used up, it repeats. For example, the following code specifies a long dash (9 pixels), a skip (3 pixels), a short dash (2 pixels) and again a skip (3 pixels). After this sequence, the pattern repeats.

```
char ldash_sdash[] = { 9, 3, 2, 3}

fl_dashedlinestyle(ldash_sdash, 4);
```

It is important to note that whenever `FL_LINE_USERDASH` is used, `fl`dashedlinestyle()+ should be called to set the dash pattern, otherwise whatever the last non-solid pattern is will be used. To use the default dash pattern, you can pass null as the `dash` parameter to `fl_dashedlinestyle()`.

By default, all lines are drawn so they overwrite the destination pixel values. It is possible to change the drawing mode so the destination pixel values play a role in the final pixel value

```
void fl_drawmode(int mode)
```

where the supported modes are

`GXcopy` The default. Overwrite. Final value = Src;

`GXxor` Boolean exclusive-or. Useful for rubber-banding. Final value: Src `xor` dest.

`GXand` Final value: Src `and` dest.

`GXor` Final value: Src `or` dest.

`GXinvert` Final value: `~`dest.

`GXnoop` Final value: dest.

To obtain the current settings of the line drawing attributes, use the following routines

```
int fl_get_linewidth(void)

int fl_get_linestyle(void)

int fl_get_drawmode(void)
```

There are also a number of high-level drawing routines available. To draw boxes the following routine exists. Almost any object class will use it to draw the bounding box of the object.

```
void fl_drw_box(int style, FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,
                   FL_COLOR col, int bw)
```

Draws a box. `style` is the type of the box, e.g `FL_DOWN_BOX`. `x`, `y`, `w`, and `h`, indicate the size of the box. `c` is the color. `bw` is the width of the boundary, which typically should be given a value `obj->bw` or `FL_BOUND_WIDTH`. Note that a negative border width indicates a "softer" up box. See `DEMOS/borderwidth.c`for the visual effect of different border widths.

There is another routine that draws a frame

```
void fl_drw_frame(int style, FL_Coord x, FL_Coord y,
                  FL_Coord w, FL_Coord h, FL_COLOR col, int bw)
```

All parameters have the usual meaning except that the frame is drawn *outside* of the bounding box specified.

To draw a slider of various types and shapes, use the following routine

```
void fl_drw_slider(int boxtype, FL_Coord x, FL_Coord y,
                   FL_Coord w, FL_Coord h,
                   FL_COLOR col1, FL_COLOR col2,
                   int slider_type,
                   double slider_size, double slider_value,
                   char *label, int parts, int inverted,
                   FL_Coord bw);
```

where `slider_type` is `FL_VERT_SLIDER` etc. See Section 17.1 for a complete list. Other parameters have the obvious meaning except for. `parts`, which can be one of the following

`FL_SLIDER_NONE` Don't draw anything.

`FL_SLIDER_BOX` Draw the bounding box only.

`FL_SLIDER_KNOB` Draw the knob only.

`FL_SLIDER_ALL` Draw the entire slider.

.

For drawing text there are two routines:

```
void fl_drw_text(int align, FL_Coord x, FL_Coord y, FL_Coord w,
                FL_Coord h, FL_COLOR c, int style, int size, char *str)

void fl_drw_text_beside(int align, FL_Coord x, FL_Coord y,
                        FL_Coord w, FL_Coord h, FL_COLOR c,
                        int style, int size, char *str)
```

where `align` is the alignment, namely, `FL_ALIGN_LEFT`, `FL_ALIGN_CENTER` etc. `x`, `y`, `w` and `h` indicate the bounding box, `c` is the color of the text, `size` is its size (in points), `style` is the style to be used (see Section 3.11.3 for valid styles), `str` is the string itself, possibly with embedded newlines it in. `fl_drw_text()` draws the text inside the bounding box according to the alignment request while `fl_drw_text_beside()` draws the text aligned outside the box. These two routines interpret a text string starting with the character `@` differently and draw some symbols instead. Note that `fl_drw_text()` shrinks the bounding box by 5 pixels on all sides before computing the alignment position.

The following routine can also be used to draw text and in addition, a cursor can optionally be drawn

```
void fl_drw_text_cursor(int align, FL_Coord x, FL_Coord y,
         FL_Coord w, FL_Coord h, FL_COLOR c, int style, int size,
         char *str, int FL_COLOR ccol, int pos)
```

where `ccol` is the color of the cursor and `pos` is the position of the cursor (-1 means show no cursor). This routine does not interpret the meta-character `@` nor does it shrink the bounding box in calculating the alignment position.

Given a bounding box and the size of an object (label or otherwise) to draw, the following routine can be used to obtain the starting position

```
void fl_get_align_xy(int align, int x, int y, int w, int h,
                     int obj_xsize, int obj_ysize,
                     int xmargin, int ymargin, int *xpos, int *ypos)
```

This routine works regardless if the object is to be drawn inside or outside of the bounding box specified by `x`,`y`,`w` and `h`.

For drawing object labels, the following routines might be more convenient

```
void fl_draw_object_label(FL_OBJECT *ob)

void fl_draw_object_label_outside(FL_OBJECT *ob)
```

These two routines assume that the alignment is relative to the full bounding box. The first routine draws the label according to the alignment, which could be inside or outside of the bounding box. The second routine will always draw the label outside of the bounding box.

An important of aspect of (re)drawing of an object is efficiency which can translate into flicker and non-responsiveness if not handled with care. For simple object like buttons or objects that do not have "movable parts", drawing efficiency is not a serious issue although you can never be too fast. For complex objects, especially those that a user can interactively change, special care should be taken.

The most important rule for efficient redrawing is don't draw it if you don't absolutely have to, regardless how simple the drawing is. Given the networking nature of X, simple or not depends not only on the host/server speed but also the connection. What this strategy entails is that the drawing should be broken into blocks and depending on the context, draw/updates only those parts that need to be updated.

Chapter 27

An example

Let us work through an example of how to create a simple object class `colorbox`. Assume we want a class with the following behavior: It should normally be red. When the user presses the mouse on it it should turn blue. When the user releases the mouse button the object should turn red again and be returned to the application program. Further, the class module should keep a total count how many times the box is pushed.

The first thing to do is to define some constants in a file `colbox.h`. This file should at least contain the class number and one or more types:

```
/* FL_USER_CLASS_START <= Class number <= FL_USER_CLASS_END */
#define FL_COLBOX        (FL_USER_CLASS_START+1)
#define FL_NORMAL_COLBOX 0     /* The only type */
```

Note that the type must start from zero onward.

Normally it should also contain some defaults for the boxtype and label alignment etc. The include file also has to declare all the functions available for this object class. I.e., it should contain:

```
extern FL_OBJECT *fl_create_colbox(int, FL_Coord, FL_Coord, FL_Coord,
                                   FL_Coord, const char *);
extern FL_OBJECT *fl_add_colbox(int, FL_Coord, FL_Coord, FL_Coord,
                                FL_Coord, const char *);
extern int fl_get_colorbox(FL_OBJECT *);
```

Secondly we have to write a module `colbox.c` that contains the different routines. First of all we need routines to create an object of the new type and to add it to the current form. We also need to have a counter that keeps track of number of times the colbox is pushed. They would look as follows:

```
typedef struct { int counter; } SPEC;   /* no. of times pushed */

FL_OBJECT *fl_create_colbox(int type, FL_Coord x, FL_Coord y,
```

```
                             FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob;

    /* create a generic object class with an appropriate ID */
    ob = fl_make_object(FL_COLBOX,type,x,y,w,h,label,handle_colbox);

    /* initialize some members */
    ob->col1 = FL_RED;
    ob->col2 = FL_BLUE;

    /* create class specific structures and initialize */
    ob->spec = fl_calloc(1, sizeof(SPEC))
    return ob;
}

FL_OBJECT *fl_add_colbox(int type, FL_Coord x, FL_Coord y,
                         FL_Coord w,  FL_Coord h, const char *label)
{
   FL_OBJECT *ob = fl_create_colbox(type,x,y,w,h,label);
   fl_add_object(fl_current_form,ob);
   return ob;
}
```

The fields `col1` and `col2` are used to store the two colors red and blue such that the user can change them when required with the routine `fl_set_object_color()`. What remains is to write the handling routine `handle_colbox()`. It has to react to three types of events: `FL_DRAW`, `FL_PUSH` and `FL_RELEASE`. Also when the box is pushed, the counter should be incremented to keep a total count. Note that whether or not the mouse is pushed on the object is indicated in the field `ob->pushed`. Hence, when pushing and releasing the mouse the only thing that needs to be done is redrawing the object. This leads to the following piece of code:

```
static int
handle_colbox(FL_OBJECT *ob, int event, FL_Coord mx, FL_Coord my,
             int key, void *xev)
{
  switch (event) {
   case FL_DRAW:
     /* Draw box */
     if (ob->pushed)
        fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col2,ob->bw);
     else
        fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col1,ob->bw);
     /* fall through */
   case FL_DRAWLABEL:
     /* Draw label */
     fl_draw_object_label(ob);
```

```
      return 0;
   case FL_PUSH:
      ((SPEC *)ob->spec)->counter++;
      fl_redraw_object(ob);
      return 0;
   case FL_RELEASE:
      fl_redraw_object(ob);
      return 1;
   case FL_FREEMEM:
      fl_free(ob->spec);
      return 0;
   }
   return 0;
}
```

That is the whole piece of code. Of course, since structure `SPEC` is invisible outside `colbox.c`, the following routine should be provided to return the total number of times the `colbox` is pushed:

```
int fl_get_colbox(FL_OBJECT *ob)
{
    if(!ob || ob->objclass != FL_COLBOX)
    {
       fprintf(stderr, "get_colbox: Bad argument or wrong type);
       return 0;
    }
    return ((SPEC *)ob->spec)->counter;
}
```

To use it, compile it into a file `colbox.o`. An application program that wants to use the new object class simply should include `colbox.h` and link `colbox.o` when compiling the program. It can then use the routine `fl_add_colbox()` to add objects of the new type to a form.

Chapter 28

New buttons

Since button-like object is one of the most important, if not *the* most important, classes in graphical user interfaces, **Forms Library** provides, in addition to the ones explained earlier, a few more routines that make create new buttons or button-like objects even easier. These routines take care of the communication between the main module and the button handler so all new button classes created using this scheme behave consistently. Within this scheme, the programmer only has to write a drawing function that draws the button. There is no need to handle events or messages from the main module and all types of buttons, radio, pushed or normal are completely taken care of by the generic button class. Further, `fl_get_button()` and `fl_set_button()` work automatically without adding any code for them.

Forms Library provides two routines to facilitate the creation of new button object classes. One of the routines, `fl_create_generic_button()`, can be used to create a generic button that has all the properties of a real button except that this generic button does not know what the real button looks like. The other routine, `fl_add_button_class()`, provide by the **Forms Library** can be used to register a drawing routine that completes the creation of a new button.

All button or button-like object has the following instance-specific structure, defined in `forms.h`, that can be used to obtain information about the current status of the button:

```
typedef struct
{
    Pixmap pixmap;           /* for bitmap/pixmap button only        */
    Pixmap mask;             /* for bitmap/pixmap button only        */
    unsigned bits_w, bits_h; /* for bitmap/pixmap button only        */
    int val;                 /* whether pushed                       */
    int mousebut;            /* mouse button that caused the push    */
    int timdel;              /* time since last touch (TOUCH buttons)*/
    int event;               /* what event triggered the redraw      */
    long cspecl;             /* for non-generic class specific data  */
    void *cspec;             /* for non-generic class specific data  */
    char *file;              /* filename for the pixmap/bitmap file  */
}
FL_BUTTON_STRUCT;
```

Of all members, only val and mousebut probably will be consulted by the drawing function. cspecl and cspecv are useful for keeping track of class status other than those supported by the generic button (e.g., you might want to add a third color to a button for whatever purposes.) These two members are neither referenced nor changed by the generic button class.

Making this structure visible somewhat breaks the **Forms Library**'s convention of hiding the instance specific data but the convenience and consistency gained by this far outweigh the compromise on data hiding.

The basic procedures in creating a new button-like object are as follows. First, just like creating any other object classes, you have to decide on a class ID, an integer between FL_USER_CLASS_START (1001) and FL_USER_CLASS_END (9999) inclusive. Then write a header file so that application programs can use this new class. The header file should include the class ID definition and function prototypes specific to this new class.

After the header file is created, you will have to write C functions that create and draw the button. Also you need an interface routine to place the newly created button onto a form.

After creating the generic button, the new button class should be made known to the button driver via the following function

```
void fl_add_button_class(int objclass,
                         void (*draw)(FL_OBJECT *ob),
                         void (*cleanup)(FL_BUTTON_SPEC *));
```

where objclass is the class ID, and draw is a function that will be called to draw the button and cleanup is a function that will be called prior to destroying the button. You need a clean-up function only if the drawing routine uses cspecv field of FL_BUTTON_SPEC to hold dynamic memory allocated by the new button.

We use two examples to show how new buttons are created. The first example is taken from the button class in the **Forms Library**, that is, real working source code that implements the button class. To illustrate the entire process of creating this class, let us call this button class FL_NBUTTON.

First we create a header file to be included in an application program that uses this button class:

```
#ifndef NBUTTON_H
#define NBUTTON_H

#define FL_NBUTTON     FL_USER_CLASS_START

extern FL_OBJECT *fl_create_nbutton(int, FL_Coord, FL_Coord,
                                    FL_Coord, FL_Coord, const char *);
extern FL_OBJECT *fl_add_nbutton(int, FL_Coord, FL_Coord,
                                 FL_Coord, FL_Coord, const char *);
#endif
```

Now the drawing function. We use obj->col1 for the normal color of the box; obj->col2 for the color of the box when pushed. We also add an extra property that when mouse moves over the button box, the box changes color. The following is the full source code that implements this:

```
typedef FL_BUTTON_STRUCT SPEC;

static void draw_nbutton(FL_OBJECT * ob)
{
    long col;

    /* box color. If pushed we use ob->col2, otherwise use ob->col1 */
    col = ((SPEC *) (ob->spec))->val ? ob->col2 : ob->col1;

    /* if mouse is on top of the button, we change the color of
     * the button to a different color. However we only do this if the
     * box has the default color.
     */
    if (ob->belowmouse && col == FL_COL1)
        col = FL_MCOL;

    /* If original button is an up_box and it is being pushed,
     * we draw a down_box. Otherwise, don't have to change
     * the boxtype
     */
    if (ob->boxtype == FL_UP_BOX && ((SPEC *) (ob->spec))->val)
       fl_drw_box(FL_DOWN_BOX, ob->x, ob->y, ob->w, ob->h, col, ob->bw);
    else
       fl_drw_box(ob->boxtype, ob->x, ob->y, ob->w, ob->h, col, ob->bw);

    /* draw the button label */
    fl_drw_object_label(ob);

    /* if the button is a return button, draw the return symbol.
     * Note that size and style are 0 as they are not used when
     * drawing symbols
     */
    if (ob->type == FL_RETURN_BUTTON)
       fl_drw_text(FL_ALIGN_CENTER, (ob->x + ob->w - 0.8 * ob->h - 1),
                   (ob->y + 0.2 * ob->h), (0.6 * ob->h),
                   (0.6 * ob->h),ob->lcol,0,0,"@returnarrow");
}
```

Note that when drawing symbols, the style and size are irrelevent and set to zero in `fl_drw_text()` above.

Since we don't use `cspecv` field, we don't have to write a clean-up function.

Next, following the standard procedures of the **Forms Library**, we code a separate routine that creates the new button[1]

[1] A separate creation routine is useful for integration into the **Form Designer**.

```
FL_OBJECT *fl_create_nbutton(int type, FL_Coord x, FL_Coord y,
                             FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob;
    ob = fl_create_generic_button(FL_NBUTTON, type, x, y,w,h, label);
    fl_add_button_class(FL_NBUTTON, draw_nbutton, 0);
    ob->col1 = FL_COL1;          /* normal color           */
    ob->col2 = FL_MCOL;          /* pushed color           */
    ob->align = FL_ALIGN_CENTER; /* button label placement */
}
```

You will also need a routine that adds the newly created button to a form

```
FL_OBJECT *fl_add_nbutton(int type, FL_Coord x, FL_Coord y,
                          FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob = fl_create_nbutton(type,x,y,w,h,label);
    fl_add_object(fl_current_form, ob);
    return ob;
}
```

This concludes the creation of button class `FL_NBUTTON`.

The next example implements a button that might be added to the **Forms Library** in the future. We call this button `crossbutton`. Normally this button shows a small up box with a label on the right. When pushed, the up box becomes a down box and a small cross appears on top of it. This kind of button obviously is best used as a push button or a radio button. However, the **Forms Library** does not enforce this. It can be enforced, however, by the application program or by object class developers.

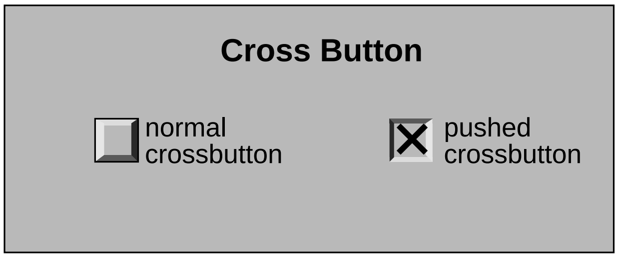

Figure 28.1: New button class

We choose to use the `ob->col1` as the color of the box and `ob->col2` as the color of the cross (remember these two colors are changeable by the application program via `fl_set_object_color()`). Note this decision on color use is somewhat arbitrary, we can easily make `ob->col2` to be the color of the button when pushed and use `ob->spec->cspecl` for the cross color (another routine `fl_set_crossbutton_crosscol(FL_OBJECT *, FL_COLOR)` should be provided to change the cross color in this case).

We start by defining the class ID and declaring the utility routine prototypes in the header file (`crossbut.h`):

```
#ifndef CROSSBUTTON_H
#define CROSSBUTTON_H

#define FL_CROSSBUTTON      (FL_USER_CLASS_START+2)

extern FL_OBJECT *fl_add_crossbutton(int, FL_Coord, FL_Coord,
                         FL_Coord, FL_Coord, const char *);

extern FL_OBJECT *fl_create_crossbutton(int, FL_Coord, FL_Coord,
                                    FL_Coord, FL_Coord, const char *);
```

Next we write the actual code that implements crossbutton class `crossbut.c`:

```
/*
 * routines implementing the "crossbutton" class
 */
#include "forms.h"
#include "crossbut.h"

typedef FL_BUTTON_STRUCT SPEC;

/** How to draw it */
static void draw_crossbutton(FL_OBJECT *ob)
{
    FL_Coord xx, yy, ww, hh;
    SPEC *sp = ob->spec;

    /* there is no visual change when mouse enters/leaves the box */
    if(sp->event == FL_ENTER || sp->event == FL_LEAVE)
        return;

    /* draw the bounding box first */
    fl_drw_box(ob->boxtype, ob->x, ob->y, ob->w, ob->h, ob->col1, ob->bw);

    /* draw the box that contains the cross */
    ww = hh = (0.5 * FL_min(ob->w, ob->h))  -1;
    xx = ob->x + FL_abs(ob->bw);
    yy = ob->y + (ob->h - hh)/2;

    /* if pushed, draw a down box with the cross */
    if(((SPEC *)ob->spec)->val)
```

```
    {
      fl_drw_box(FL_DOWN_BOX,xx,yy,ww,hh,ob->col1,ob->bw);
      fl_drw_text(FL_ALIGN_CENTER, xx-2, yy-2, ww+4, hh+4, ob->col2,
                  0, 0, "@9plus");
    }
    else
      fl_drw_box(FL_UP_BOX,xx,yy,ww,hh,ob->col1, ob->bw);

    /* label */
    if (ob->align == FL_ALIGN_CENTER)
        fl_drw_text(FL_ALIGN_LEFT, xx + ww + 2, ob->y, 0, ob->h,
                    ob->lcol, ob->lstyle, ob->lsize, ob->label);
    else
        fl_draw_object_label_outside(ob);

    if (ob->type == FL_RETURN_BUTTON)
       fl_drw_text(FL_ALIGN_CENTER,
                (FL_Coord)(ob->x + ob->w - 0.8 * ob->h),
                (FL_Coord)(ob->y + 0.2 * ob->h),
                (FL_Coord)(0.6 * ob->h),
                (FL_Coord)(0.6 * ob->h),ob->lcol,0,0,"@returnarrow");
}
```

This button class is somewhat different from the normal button class (FL_BUTTON) in that we enforce the appearance of a crossbutton so that an un-pushed crossbutton always has an upbox and a pushed one always has a downbox. Note that the box that contains the cross is *not* the bounding box of a crossbutton although it can be if the drawing function is coded so.

Rest of the code simply takes care of interfaces:

```
/* creation routine */
FL_OBJECT *
fl_create_crossbutton(int type, FL_Coord x, FL_Coord y, FL_Coord w,
                      FL_Coord h, const char *label)
{
     FL_OBJECT *ob;
     fl_add_button_class(FL_CROSSBUTTON, draw_crossbutton, 0);

     /* if you want to make cross button only available for
      * push or radio buttons, do it here as follows:
        if(type != FL_PUSH_BUTTON && type != FL_RADIO_BUTTON)
           type = FL_PUSH_BUTTON;
      */
     ob = fl_create_generic_button(FL_CROSSBUTTON,type,x,y,w,h,label);
     ob->boxtype = FL_NO_BOX;
     ob->col2 = FL_BLACK;   /* cross color */
     return ob;
```

```
}


/* interface routine to add a crossbutton to a form */
FL_OBJECT *
fl_add_crossbutton(int type, FL_Coord x, FL_Coord y, FL_Coord w,
                   FL_Coord h, const char *label)
{
    FL_OBJECT *ob = fl_create_crossbutton(type, x, y, w, h, label);
    fl_add_object(fl_current_form, ob);
    return ob;
}
```

The actual code is in `DEMOS/crossbut.c` and `DEMOS/crossbut.h`. An application program only needs to `#include` the header file `crossbut.h` and link with `crossbut.o` to use this new object class. There is no need to change or re-compile the **Forms Library**. Of course, if you really like the new object class, you can modify the system header file `forms.h` to include your new class header file automatically (either through inclusion at compile time or include the actual header). You can also place the object file (`crossbut.o`) in `libforms.a` if you wish. Note however, library so created may *not* be distributed.

Since the current version of **Form Designer** does not support any new object classes developed as outlined above, the best approach is to use another object class as stubs when creating a form, for example, you might want to use `checkbutton` as stubs for `crossbutton`. Once the position and size are satisfactory, generate the C-code and then manually change `checkbutton` to `crossbutton`. You probably can automate this with some scripts.

Finally there is a demo program utilizing this new button class. The program is in `newbutton.c`.

Chapter 29

Using a pre-emptive handler

Pre-emptive handlers come into being due to reasons not related to developing new classes. They are provided for the application programs to have access to the current state or event of a particular object. However, with some care, this preemptive handler can be used to override parts of the original built-in handler thus yielding a new class of objects.

29.1 The Pre-emptive and Post Object Handler

As mentioned earlier, an object module communicates with the main module via events and the agent is the event handler, which determines how an object responds to various events such as a mouse click or a key press. A pre-emptive handler is a handler which, if installed, gets called first by the main module when an event for the object occurs. The pre-emptive handler has the option to override the built-in handler by informing the main module not to call the built-in handler, thus altering the behavior of the built-in objects. The post handler, on the other hand, is called when the object handler has finished its tasks and thus does not offer the capability of overriding the built-in handler. It is much safer, however.

The API to install a pre- or post-handler for an object is as follows

```
typedef int (*FL_HANDLEPTR)(FL_OBJECT *ob, int event,
                            FL_Coord mx, FL_Coord my,
                            int key, void *raw_event);

void fl_set_object_prehandler(FL_OBJECT *ob, FL_HANDLEPTR phandler);
void fl_set_object_posthandler(FL_OBJECT *ob, FL_HANDLEPTR phandler);
```

Where `event` is the generic event in the **Forms Library**, that is, `FL_DRAW`, `FL_ENTER` etc. Parameter `mx, my` are the mouse position and `key` is the key pressed. The last parameter `raw_event` is the (cast) XEvent that caused the invocation of the pre- or post-handler. Again, not all `FL` event has corresponding `xev` and any dereferencing of `xev` should only be done after making sure it is not null.

Notice that the pre- and post-handler have the same function prototype as the built-in handler. Actually they are called with exactly the same parameters by the event dispatcher. The prehandler should return !FL_PREEMPT if the processing by the built-in handler should continue. A return value of FL_PREEMPT will prevent the dispatcher from calling the built-in handler. The post-handler is free to return anything and the return value is not used. Note that a post-handler will receive all events even if the object the post-handler is registered for does not. For example, a post-handler for a box (a static object that only receives FL_DRAW) receives all events.

See demo program preemptive.c and xyplotall.c for examples.

Bear in mind that modifying the built-in behavior is in general not a good idea. Using the pre-emptive handler for the purpose of “peeking", however, is quite legitimate and can be useful in some situations.

Part V

Appendices

Appendix A

Overview of main routines

In this appendix we give a brief overview of all main routines that are available. For an overview of all routines related to specific object classes see Part III.

A.1 Version Information

The header file, `forms.h`, defines three symbolic constants which you can use to conditionally compile your application. The three symbolic constants are

`FL_VERSION`	The major version number.
`FL_REVISION`	Revision number.
`FL_INCLUDE_VERSION`	Derived as `FL_VERSION` $\times$ 1000 + `FL_REVISION`

There is also a routine that can be used to obtain the library version at run time:

```
int fl_library_version(int *version, int *revision)
```

The function returns a consolidated version information, computed as `version` $\times 1000$ + `revision`. For example, for library version 1 revision 21 (1.21), the function returns a value of 1021 with `version` and `revision` (if not null) set to 1 and 21 respectively.

It is always a good idea to check if the header and the run time library are of the same version and take appropriate actions when they are not. This is especially important for `version` < 1.

To obtain the version number of the library used in an executable, run the command with `-flversion` option, which will print the complete version information.

A.2 Initialization

The routine

```
Display *fl_initialize(int *argc, char *argv[], const char *appclass,
                       XrmOptionDescList app_opt, int n_app_opt)
```

initializes the **Forms Library**and returns a pointer to the `Display` structure if a connection is made otherwise a null is returned. This function should always be called before any other calls to the **Forms Library** are made (except `fl_set_defaults()` and a few other functions that alter some of the defaults of the library). The meaning of the arguments are as follows

`argc,argv` Command line parameters. The application name is derived from `argv[0]` by stripping leading path names and trailing period and extension, if any. Due to the way the X resources (and command line argument parsing) work, the executable name should not contain `.` or `*`.

`appclass` The application class name, which typically is the generic name for all instances of this application. If no meaningful class name exists, it is typically given (or converted to if non given) as the application name with the first letter capitalized (second if the first letter is an X).

`app_opt` Specifies how to parse the application-specific resources.

`n_app_opt` Number of entries in the option list.

The `fl_initialize` function builds the resource database, calls Xlib *XrmParseCommand*(3X11) function to parse the command line, and performs other per display initialization.

All recognized options are removed from the argument list and their corresponding values set. **Forms Library** provides appropriate defaults for all options. The following are the defaults:

Options	Value type	Meaning	Default
`-debug` *level*	int	prints debug information	0
`-name` *appname*	string	changes application name	none
`-flversion`	none	prints the version of the library	false
`-sync`	none	requests synchronous mode(debug)	false
`-display` *host:dpy*	string	specifies remote host	$DISPLAY
`-visual` *class*	string	TrueColor, PseudoColor ...	best
`-depth` *depth*	int	specifies prefered visual depth.	best
`-vid` *id*	long	specifies prefered visual ID	
`-private`	none	forces private colormap.	false
`-shared`	none	forces shared colormap.	false
`-stdcmap`	none	forces standard colormap.	false
`-double`	none	enables double buffering	false
`-bw` *width*	int	changes border width	3
`-rgamma` *gamma*	float	specifies red gamma	1.0
`-ggamma` *gamma*	float	specifies green gamma	1.0
`-bgamma` *gamma*	float	specifies blue gamma	1.0

“best" in the above table means the visual that has the most colors, which may or may not be the server default. There is a special command option `-visual Default` that sets both the visual and depth to the X server default. If a visual ID is requested, it overrides depth or visual if specified. Visual Id can also be requested programmatically (before `fl_initialize`) via the following function

```
void fl_set_visualID(long id)
```

Note that all command line options can be abbreviated, thus if the application program uses single character options, they might clash with the built-ins. For example, if you use `-g` as a command line option to indicate geometry, it might not work as `-g` matches `-ggamma` in the absence of `-ggamma`. Thus you should avoid using single character command line options.

If border width is set to a negative number, all objects appear to be softer and some people might prefer `bw -2`..

Depending on your application, **XForms** defaults may or may not be appropriate. E.g., on machines capable of 24bits visuals, **Forms Library** always selects the deeper 24bits visual. If your application only uses a limited number of colors, it would typically be faster if a visual other than 24bits is selected.

There are a couple of ways to override the default settings. You can provide an application specific resource database distributed with your program. The easiest way, however, is to set up your own program default programmatically without affecting the users' ability to override with command line options. For this, you can use the following routine *before* `fl_initialize()`:

```
void fl_set_defaults(unsigned long mask, FL_IOPT *flopt)
```

In addition to setting a preferred visual, this function can also be used to set other program defaults, such as label font size, unit of measure for form sizes etc.

See Table A.1 for a list of the masks and the members of `FL_IOPT`.

A special visual designation, `FL_DefaultVisual` and command line option equivalent `-visual Default` are provided to set the program default to the server's default visual class and depth.

If you set up your resource specifications to use class names instead of instance names, users can then list instance resources under arbitrary name that is specified with the *-name* option.

Coordinate units can be in pixels, points (1/72 inch), mm (milli-meters), cp (centi-point, i.e., 1/100 of a point) or cmm (centi-millimeter). The pre-defined designations (enums) for `coordUnit` are `FL_COORD_PIXEL`, `FL_COORD_POINT`, `FL_COORD_MM`, `FL_COORD_centiPOINT`, and `FL_COORD_centiMM`. `coordUnit` can be changed anytime, but typically you would do this prior to creating a form, presumably to make the size of the form screen resolution independent. The basic steps in doing this may look something like the following:

```
int oldcoordUnit = fl_get_coordunit();
fl_set_coordunit(FL_COORD_POINT);
fl_bgn_form(...);
/* add more objects */
fl_end_form();
fl_set_coordunit(oldcoordunit);
```

As you can see, convenience functions `fl_set_coordunit()` and `fl_get_coordunit()` are provided to change the unit of measure.

Structure	Mask Name	Meaning
typedef struct {		
int debug;	FL_PDDebug	Debug level (0-5)
int depth;	FL_PDDepth	Preferred visual depth
int vclass;	FL_PDVisual	Preferred visual. TrueColor etc
int doubleBuffer	FL_PDDouble	Simulate double buffering
int buttonFontSize	FL_PDButtonFontSize	Default Button label fontsize
int menuFontSize	FL_PDMenuFontSize	Menu label fontsize
int choiceFontSize	FL_PDChoiceFontSize	Choice label and choice text fontsize
int browserFontSize	FL_PDBrowserFontSize	Browser label and text fontsize
int inputFontSize	FL_PDInputFontSize	Input label and text fontsize
int labelFontSize	FL_PDLabelFontSize	label fontsize for all other objects (box, pixmap etc.)
int pupFontSize	FL_PDPupFontSize	Fontsize for pop-ups
int privateColormap	FL_PDPrivateMap	Select private colormap if appropriate
int sharedColormap	FL_PDSharedMap	Force shared colormap always
int standardColormap	FL_PDStandardMap	Force standard colormap
int scrollbarType	FL_PDScrollbarType	Scrollbar for browser and input
int ulThickness	FL_PDULThickness	Underline thickness
int ulPropWidth	FL_PDULPropWidth	Underline width. 0 for const. width
int backingStore	FL_PDBS	turn BackingStore on and off
int coordUnit	FL_PDCoordUnit	Unit of measure: pixel, mm, point
int borderWidth	FL_PDBorderWidth	Height of an object
} FL_IOPT;		

Table A.1: `FL_IOPT` structure

Some of the defaults are "magic" in that their exact values depend on the context or platform. For example, the underline thickness by default is 1 for normal font and 2 for bold font.

There exists a convenience function to set the application default border width

```
void fl_set_border_width(int border_width)
```

which is equivalent to

```
FL_IOPT fl_cntl;
fl_cntl.borderWidth = border_width;
fl_set_defaults(FL_PDBorderWidth, &fl_cntl);
```

Typically this function, if used, should appear before `fl_initialize()` so the user has the option to override the default via resource or command line options. Note that this function that not affect the popup border width, which is controlled by `fl_setpup_default_bw()`.

To change the default scrollbars (which are `THIN_SCROLLBAR`s) used in browser and input object, the following convenience function can be used:

```
void fl_set_scrollbar_type(int type)
```

where `type` can be one of the following

`FL_NORMAL_SCROLLBAR` The basic scrollbar.

`FL_THIN_SCROLLBAR` The thin scrollbar

`FL_NICE_SCROLLBAR` The nice scrollbar

`FL_PLAIN_SCROLLBAR` Similar to thin scrollbar, but not as fancy.

which is equivalent to

```
FL_IOPT fl_cntl;
fl_cntl.scrollbarType = type;
fl_set_defaults(FL_PDScrollbarType, &fl_cntl);
```

It is recommended that this function be used before `fl_initialize()` so the user has the option to override the default through application resources.

Prior to version V0.80, the origin of **XForms**'s coordinate system was at the lower-left corner of the form. The new **Form Designer** will convert the form definition file to the new coordinate system, i.e., origin at the upper-left, so no manual intervention is required. To help those who lost the .fd files or otherwise can't use the new fdesign, a compatibility function is provided

```
void fl_flip_yorigin(void)
```

Note however, this function must be called prior to `fl_initialize` and is a no-op after that.

For proportional font, substituting tabs with spaces is not always appropriate because this most likely will fail to align text properly. Instead, a tab is treated as an absolute measure of distance, in pixels, and a tab stop will always end at multiples of this distance. Application program can adjust this distance by setting the tab stops using the following routine

```
void fl_set_tabstop(const char *s)
```

where `s` is a string whose width in pixels is to be used as the tab length. The font used to calculate the width is the same font that is used to render the string in which the tab is embedded. The default is `s = "aaaaaaaa"`, i.e., eight `'a'`s;

Before we proceed further, some comments about double buffering are in order. Since Xlib does not support double buffering, **Forms Library** simulates this functionality with pixmap bit-blting. In practice, the effect is hardly distinguishable from double buffering and performance is on par with multi-buffering extensions (It is slower than drawing into a window directly on most workstations however). Bear in mind that pixmap can be resource hungry, so use this option with discretion.

In addition to using double buffering throughout an application, it is also possible to use double buffering on a per-form or per-object basis by using the following routines:

```
void fl_set_form_dblbuffer(FL_FORM *form, int yes)

void fl_set_object_dblbuffer(FL_OBJECT *obj, int yes)
```

Currently double buffering for objects having a non-rectangular box might not work well. A non-rectangular box means that there are regions within the bounding box that should not be painted, which is not easily done without complex and expensive clipping and unacceptable inefficiency. XForms gets around this by painting these regions with the form's backface color. In most cases, this should prove to be adequate. If needed, you can modify the background of the pixamp by changing `obj->dbl_background` after switching to double buffer.

Normally the **Forms Library** reports errors to stderr. This can be avoided or modified by registering an error handling function

```
void fl_set_error_handler(void (*user_handler)
                          (const char *where, const char *fmt,...))
```

The library will call the `user_handler` with a string indicating where or which function an error occured, and a formatting string (see *sprintf*3) followed by zero or more arguments. To restore the default handler, set `user_handler` to null. You can call this function anytime or as many times as you wish.

You can also instruct the default message handler to log the error to a file instead of printing to stderr

```
void fl_set_error_logfp(FILE *fp)
```

For example, `fl_set_error_logfp(fopen("/dev/null","w"))` turns off the default error reporting to stderr.

For some error messages, in addition to being printed to stderr, a dialog box will be shown that requires actions from the user. To turn this off and on, the following routine is available

```
void fl_show_errors(int show)
```

`show` indicates whether to show (1) or not show (0) the errors.

The fonts used in all forms can be changed using the routine

```
void fl_set_font_name(int numb,const char *name)
```

where `numb` is a number between 0 and (`FL_MAXFONTS-1`).

See section 3.11.3 for details. A redraw of all forms is required to actually see the change for visible forms.

Since the dimension of an object is typically given in pixels, depending on the server resolution and the font used, this can lead to unsatisfactory user interfaces. For example, a button designed to (just) contain a label in a 10pt font on a 75 DPI monitor will have the label overflow the button on a 100DPI monitor. This comes about because a character in a 10pt font with 75DPI resolution may have 10 pixels while the same character in the same 10 pt font with 100DPI resolution may have 14 pixels. Thus when designing the interfaces, leave a few pixels extra for the object. Or use a resolution independent unit, such as point, or centi-point etc.

Using a resolution independent unit for the object size should solve the font problems theoretically. In practice, this approach may still prove to be vulnerable. The reason is the discreteness of both the font resolution and the monitor/server resolutions. The standard X fonts only come in two discrete resolutions, 75 DPI and 100 DPI. Due to the variations in monitor resolutions, the same theoretically same sized font, say a 10pt font, can vary in sizes (pixels) up to 30% depending on the server (rendering a font on a 80DPI monitor will cause error in sizes regardless if 75 or 100DPI font is used.) This has not even taken into account the fact that a surprising number of systems have wrong font paths (e.g., a 90DPI monitor using 75DPI fonts etc).

With the theoretical and practical problems associated with X fonts, it is not practical for **XForms** to hard-code default font resolution and it is not practical to use the resolution information obtained from the server either as information obtained from the server regarding monitor resolution is highly unreliable. Thus, **XForms** does not insist on using fonts with specific resolutions and instead it leaves the freedom to select the default fonts of appropriate resolutions to the system administrators.

Given all these uncertainties regarding fonts, as a workaround, **XForms** provides a function that can be used to adjust the object size dynamically according to the actual fonts loaded:

```
double fl_adjust_form_size(FL_FORM *form)
```

This function works by computing the size (in pixels) of every object on the form that has an inside label and comparing it to the size of the object, scaling factors are computed if any object's label does not fit. The maximum scaling factor found are used to scale the form so every object label fits inside the object. It will never shrink a form. The function returns the overall scaling factor. In scaling the form, the aspect ratio of the form is kept and all object gravity specifications are ignored. Since this function is meant to compensate for font size and server display resolution variations, scaling is limited to 125% per invocation. The best place to use this function is right after the creation of the forms. If the forms are properly designed, this function should be a no-op on the machine the forms are designed. **Form Designer** has a special flag *-compensate* and resource `compensate` to request the emission of this function automatically for every form created. It is likely that this will become the default once the usefulness of it is established.

There is a similar function that works the same way, but on an object-by-object basis and further it allows explicit margin specifications:

```
void fl_fit_object_label(FL_OBJECT *obj, FL_Coord hm, FL_Coord vm);
```

where `hm` and `vm` are, respectively, the horizontal and vertical margins to leave on each side of the object. This function works by computing the object label size and comparing it to the object size. If the label does not fit inside the object with the given margin, the entire form the object is on is scaled so the object label fits. In scaling the form, all gravity specification is ignored but the aspect ratio of the form (thus of objects) is kept. This function will not shrink a form. You can use this function on as many objects as you choose. Of course the object has to have a label inside the object for this function to work.

In some situations **Forms Library** may modify some of the server defaults. All modified defaults are restored as early as possible by the main loop and in general when an application exits, all server defaults are restored. The only exception is that when exiting from a callback that is activated by shortcuts. Thus it is recommended that the cleanup routine `fl_finish()` be called prior to exiting an application or register it via *atexit*(3)

```
void fl_finish(void)
```

In addition to restoring all server defaults, `fl_finish()` will also shut down the connection.

A.3 Creating forms

```
FL_FORM *fl_bgn_form(int type,FL_Coord w,FL_Coord h)
```

Starts the definition of a form. `type` is the type of the box that is used as a background. `w` and `h` give the width and height of the form. The function returns a pointer to the form created.

```
void fl_end_form()
```

End the definition of a form.. Between these two calls, various objects, including group of objects, are added to the form.

```
FL_OBJECT *fl_bgn_group()
```

Begin the definition of a group of objects inside the form. It returns a pointer to the group. Groups should never be nested.

```
FL_OBJECT * fl_end_group(void)
```

Ends the definition of a group.

Groups are useful for two reasons. First of all, it is possible to hide or deactivate groups of objects. This is often very handy to dynamically change the appearance of a form depending on the context or selected options. In addition, it can also be used as a shortcut to set some particular attributes of several objects. It is not uncommon that you want several objects to maintain their relative positioning upon form resizing. This requires to set the gravity for each object. If these objects are placed inside a group, setting the gravity attributes of the group would suffice.

The second reason for using groups is for radio buttons. Radio buttons are considered related only if they belong to the same group. Using groups is the only way to place unrelated groups of radio buttons on a single form without interference from each other.

```
void fl_addto_group(FL_OBJECT *group)
```

reopens a group for adding more objects to it. Any new objects added are appended at the end of the group.

```
void fl_addto_form(FL_FORM *form)
```

Reopens a form for adding objects to it.

```
void fl_delete_object(FL_OBJECT *obj)
```

Removes an object from the form it is in.

```
void fl_free_object(FL_OBJECT *obj)
```

Frees the memory for an object. (Object should be deleted first.) An object after being freed should not be referenced.

```
void fl_free_form(FL_FORM *form)
```

Frees the memory for a form, together with all its objects. The form should not be visible.

A.4 Setting attributes

A number of general routines are available for setting attributes. Unless stated otherwise, all attributes altering routines affect the appearance or geometry of the object immediately if the object is visible.

```
void fl_set_object_color(FL_OBJECT *obj, int col1, int col2)
```

Sets the two colors that influence the appearance of the object.

attributes:

```
void fl_set_object_boxtype(FL_OBJECT *obj, int boxtype)
```

Changes the shape of the bounding box of the object.

There is also a function to change the border width of an object

```
fl_set_object_bw(FL_OBJECT *obj, int bw)
```

If the requested border width is 0, -1 is used.

```
void fl_set_object_position(FL_OBJECT *ob, FL_Coord x, FL_Cood y)
```

sets a new position for the object. If the object is visible, it is moved to the new location.

```
void fl_set_object_size(FL_OBJECT *ob, FL_Coord w, FL_Coord h)
```

changes the object size while keeping the upper-left corner of the bounding box unchanged.

```
void fl_set_object_geometry(FL_OBJECT *ob, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h)
```

sets both the position of the size of an object.

To obtain the object geometry, use the following routines

```
void fl_get_object_geometry(FL_OBJECT *ob, FL_Coord *x, FL_Coord *y,
                            FL_Coord *w, FL_Coord *h)

void fl_get_object_bbox(FL_OBJECT *ob, FL_Coord *x, FL_Coord *y,
                        FL_Coord *w, FL_Coord *h)
```

The difference between these two functions is that `fl_get_object_bbox()` returns the bounding box size that has the label size figured in.

Some objects in the library are composite objects that consist of other objects. For example, the scrollbar object is made of a slider and two scroll buttons. To get a handle to one of the components of the composite object, the following routine is available

```
FL_OBJECT *fl_get_object_component(FL_OBJECT *ob, int objclass,
                                   int type, int number)
```

where `ob` is the composite object; `objclass` and `type` are the component object's class ID and type; and `number` is the sequence number of the desired object in case the composite has more than one object of the same class and type. You can use a constant -1 for `type` to indicate any type of class `objclass`. Function returns the object handle if the requested object is found otherwise 0. For example, to obtain the object handle to the horizontail scrollbar in a browser, code similiar to the following can be used

```
hscrollbar = fl_get_object_component(browser, FL_SCROLLBAR,
                                     FL_HOR_THIN_SCROLLBAR, 0)
```

```
void fl_set_object_lcol(FL_OBJECT *obj, int lcol)

void fl_set_object_lsize(FL_OBJECT *obj, int lsize)

void fl_set_object_lstyle(FL_OBJECT *obj, int lstyle)

void fl_set_object_lalign(FL_OBJECT *obj, int align)

void fl_set_object_label(FL_OBJECT *obj, const char *label)
```

These routines set the color, size, style, alignment and text of the label of the object.

```
void fl_set_object_resize(FL_OBJECT *obj, unsigned howresize)

void fl_set_object_gravity(FL_OBJECT *obj,
                           unsigned NWgravity, unsigned SEgravity)
```

If you change many attributes of a single object or many objects in a visible form, the changed object is redrawn after each change. To avoid this, put the changes between calls to

```
void fl_freeze_form(FL_FORM *form)
```

and

```
void fl_unfreeze_form(FL_FORM *form)
```

There are also routines that influence the way events are dispatched. These routines are provided mainly to facilitate the development of (unusual) new objects where attributes need to be changed on the fly. These routines should not be used on the built-in ones.

To enable or disable an object to receive the `FL_STEP` event, use the following routine

```
void fl_set_object_automatic(FL_OBJECT *obj, int flag)
```

To enable or disable an object to receive the `FL_DBLCLICK` event, use the following routine

```
void fl_set_object_dblclick(FL_OBJECT *obj, int timeout)
```

where `timeout` (in milli-seconds) specifies the maximum time interval between two clicks to be considered a double-click (0 disables double-click detection).

```
void fl_show_object(FL_OBJECT *obj)
```

Makes the object, or the group if `obj` is a group, visible.

```
void fl_hide_object(FL_OBJECT *obj)
```

makes the object or group invisible.

```
void fl_trigger_object(FL_OBJECT *obj);
```

returns `obj` to the application program or calls `obj`'s callback if one exists.

```
void fl_set_focus_object(FL_FORM *form,FL_OBJECT *obj)
```

Set the input focus in form `form` onto object `obj`.

Note however, if this routine is used as a response to an `FL_UNFOCUS` event, i.e., as an attempt to override the focus assignment by the main loop from within an object event handler, this routine will not work as the main loop assigns a new focus object upon return from the object event handler, which undoes the focus change inside the event handler. To override and only when overriding the `FL_UNFOCUS` event, the following routine should be used:

```
void fl_reset_focus_object(FL_OBJECT *obj)
```

Use the following routine to obtain the object that has the focus on a form

```
FL_OBJECT *fl_get_focus_object(FL_FORM *form)
```

The routine

```
void fl_set_object_callback(FL_OBJECT *obj,
                            void (*callback)(FL_OBJECT *, long),
                            long argument)
```

binds a callback routine to an object.

To invoke the callback manually (as opposed to invocation by the main loop), use the following function

```
void fl_call_object_callback(FL_OBJECT *obj)
```

If the object `obj` does not have a callback associated with it, this call has not effect.

```
void fl_set_form_callback(FL_FORM *form,
                   void (*callback)(FL_OBJECT *, void *), void *data)
```

Binds a callback routine to an entire form.

It is sometimes useful to obtain the last event from within a callback function, e.g., to implement different functionalities depending on which button triggers the callback. For this, the following routine can be used from within a callback function.

```
const XEvent *fl_last_event(void)
```

Sometimes, it may be desirable to obtain hardcopies of some objects in a what-you-see-is-what-you-get (WYSISYG) way, especially those that are dynamic and of vector-graphics in nature. To this end, the following routine exists:

```
int fl_object_ps_dump(FL_OBJECT *ob, const char *fname);
```

The function will output the specified object in POSTSCRIPT. If `fname` is null, the fselector will be called to prompt the file name from the user. The function returns a negative number if no output is generated due to error conditions. At the moment, only `FL_XYPLOT` object is supported.

The object must be visible at the time of the function call. The hardcopy should mostly be WYSIWYG and centered on the printed page. The orientation is determined such that a balanced margin results, i.e., if the width of the object is larger than the height, the landscape mode will be used. Further, if the object is too big to fit on the printed page, a scale factor will be applied so the object fits. Also the box underneath the object is by default no drawn and in the default black&white mode, all curves are drawn in black. See demo `xyplotover.c` for an example output.

It is possible to customerize the output by changing the postscript output control parameters via the following routine:

```
FLPS_CONTROL *flps_init(void)
```

The typical use is to call this routine to obtain a handle to the postscript output control structure and change the control structure members to suit your need before calling `fl_object_ps_dump()`. You should not free the returned buffer.

The control structure has the following members

`ps_color` The choices are full color (`FLPS_COLOR`), grayscale (`FLPS_GRAYSCALE`), and black&white (`FLPS_BW`). The default for xyplot is black and white. In this mode, all drawings are black, on a white background. If `drawbox` (see below) is true, drawing color can be either white or black depending on the specified color.

`orientation` The valid choices are `FLPS_AUTO`, `FLPS_PORTRAIT` and `FLPS_LANDSCAPE`. The default is `FLPS_AUTO`.

`auto_fit` By default, this is true so the object always fits the printed page. Set it to false (0) to turn auto-scaling off.

`eps` Either 0 or 1.

`drawbox` Set this to 1 if the box of the object is to be drawn.

`xdpi,ydpi` These two are the screen resolution. The default is to use the actual resolution of the display. Note by setting a dpi number smaller or larger than the actual resolution, the output object is in effect being enlarged or shrunk.

`paper_w` The paper width, in inches. The default is 8.5in.

`paper_h` The paper height, in inches. The default is 11in.

To generate a POSTSCRIPT output of a form or forms, use the `fd2ps` program documented in Chapter 13.

A.5 Doing interaction

```
long fl_show_form(FL_FORM *form,int place,int border,const char *title)
```

Displays a form on the screen. `place` controls the position and size of the form. `border` indicates whether a border (window manager's decoration) should be drawn around the form. In this case `title` is the name of the window and its associated icon if any. The routine returns the window identifier of the form. For resource and identification purposes, the form name is taken to be the title with space removed and the first character lower-cased. E.g., if a form has a title `Foo Bar` the form name is derived as `fooBar`.

There are variations on the `border` requests:

`FL_FULLBORDER`	full border with title showing.
`FL_TRANSIENT`	border possibly with less decoration.
`FL_NOBORDER`	no border at all.

Multiple forms can be displayed at the same moment.

Note that `FL_NOBORDER` might have adverse effect on keyboard focus and is not very friendly to other applications (it is close to impossible to move a form that has no border). Thus use this feature with discretion. The only situation where `FL_NOBORDER` is appropriate is for automated demonstration suites or when the application program *must* obtain an input or a mouse click from the user, and even then all other forms should be deactivated while a borderless form is active. For almost all situations where the application must demand an action from the user, `FL_TRANSIENT` is preferred. Also note you can't iconify a form that has no border and under most window managers, `FL_TRANSIENT` form can't be iconified either.

On additional property (under almost all window managers) of a transient window is that it will stay on top of the main form, which the application program can designate using

```
void fl_set_app_mainform(FL_FORM *form)
```

By default, the main form is set automatically by the library to the first full-bordered form shown.

To obtain the current main form, use the following routine

```
FL_FORM *fl_get_app_mainform(void)
```

In some situations, either because the concept of an application main form does not apply (for example, an application might have multiple full-bordered windows), or under some (buggy) window managers, the designation of a main form may cause stacking order problems. To workaround these, the following routine can be used to disable the designation of a main form (before any full-bordered form is shown)

```
void fl_set_app_nomainform(int flag)
```

with a true `flag`

All visible forms will have the following properties set

```
WM_CLASS
WM_CLIENT_MACHINE
WM_NAME
```

In addition, the first full border form will have the `WM_COMMAND` property set and is by default the application main form.

The application program can raise a form to the top of the screen so no other forms obscure it by using the following routine

```
void fl_raise_form(FL_FORM *form)
```

Similar routine exists that lowers a form to the bottom of the stack

```
void fl_lower_form(FL_FORM *form)
```

When placing a form on the screen using place `FL_PLACE_GEOMETRY` the position and size can be set using the routines

```
void fl_set_form_position(FL_FORM *form, FL_Coord x, FL_Coord y)


void fl_set_form_size(FL_FORM *form, FL_Coord w, FL_Coord h)

void fl_scale_form(FL_FORM *form, double xsc, double ysc)
```

Where the last routine scales with a factor with respect to the current size. These routines can also be used when the form is visible.

If interactive resizing is to be allowed, (e.g., form shown using `FL_PLACE_FREE`), it can be useful to limit the range the size of a form can take. To this end, the following functions are available

```
void fl_set_form_minsize(FL_FORM *form, FL_Coord minw, FL_Coord minh);

void fl_set_form_maxsize(FL_FORM *form, FL_Coord maxw, FL_Coord maxh);
```

Although these two routines can be used before or after a form becomes visible, not all window managers honor such requests once the window is visible. Also note that these constraints routines only applies to the next `fl_show_form()`.

To set or change the icon shown when a form is iconified, use the following routine

```
void fl_set_form_icon(FL_FORM *form, Pixmap icon, Pixmap mask)
```

where `icon` can be any valid Pixmap ID. (See Sections 15.5 and 15.6 for some of the routines that can be used to create Pixmaps) Note that the previous icon if not freed or modified in anyway.

If for any reason, you would like to change the form title after it is shown, the following call can be used (this will also change the icon title)

```
void fl_set_form_title(FL_FORM *form, const char *name)
```

The routine

```
void fl_hide_form(FL_FORM *form)
```

hides the particular form, i.e., closes its window and all subwindows,

To check if a form is visible or not, the following can be used

```
int fl_form_is_visible(FL_FORM *form)
```

```
FL_OBJECT *fl_do_forms(void)
```

Does the interaction with the currently displayed forms. The routine ends when the state of some object changes. A pointer to this object is returned if no callback is bound to it.

```
FL_OBJECT *fl_check_forms(void)
```

Second way of doing interaction with the currently displayed forms. The routine returns immediately `NULL` unless the state of some object changes in which case a pointer to this object is returned.

```
FL_OBJECT *fl_do_only_forms(void)
```

```
FL_OBJECT *fl_check_only_forms(void)
```

Same as `fl_{do|check}_forms` except that these routines do not handle user events generated by application windows via `fl_winopen()` or similar routines.

```
void fl_activate_form(FL_FORM *form)
```

Activates a form for user interaction.

```
void fl_deactivate_form(FL_FORM *form)
```

Deactivates a form to stop user interaction with it.

```
void fl_deactivate_all_forms(void)

void fl_activate_all_forms(void)
```

Activates or deactivates all forms to stop user interaction with them.

You can also register for a form callbacks that are invoked whenever the activation status of the form is changed:

```
typedef void (*FL_FORM_ATACTIVATE)(FL_FORM *, void *);

FL_FORM_ACTIVATE fl_set_form_atactivate(FL_FORM *form,
                        FL_FORM_ATACTIVATE callback, void *data);

FL_FORM_ACTIVATE fl_set_form_atdeactivate(FL_FORM *form,
                        FL_FORM_ATACTIVATE callback, void *data);
```

```
void fl_activate_object(FL_OBJECT *obj)
```

Activates an object for user interaction.

```
void fl_deactivate_object(FL_OBJECT *obj)
```

Deactivates an object to stop user interaction with it.

```
void fl_redraw_object(FL_OBJECT *obj)
```

This routine redraws the particular object. If `obj` is a group it redraws the complete group. Normally you should never need this routine because all library routines take care of redrawing but there might be situations in which a redraw is required.

```
void fl_redraw_form(FL_FORM *form)
```

Redraws an entire form.

For non-form windows, i.e., those created with `fl_winopen()` or similar routines by the application program, the following means of interaction are provided (note that these do not work on form windows, for which a different set of functions exist. See Section D for details)

```
void fl_set_event_callback(void (*callback)(void *xevent, void *data))
```

Sets up a callback routine for all user events.

It is possible to set up callback functions on a per window/event basis using the following routines

```
typedef int (*FL_APPEVENT_CB)(XEvent *xev, void *user_data);

FL_APPEVENT_CB fl_add_event_callback(Window win, int xevent_type,
                           FL_APPEVENT_CB callback, void *user_data);

void fl_remove_event_callback(Window win, int xevent_type)
```

These functions manipulate event callback functions for the window specified and will be called when `xevent_type` is pending for the window. If `xevent_type` is zero, it signifies a callback for *all* event for window `win`. Note that **Forms Library** does not solicit any event for the caller, i.e., **Forms Library** assumes the caller opens the window and solicits all events before calling these routines. To let **Forms Library** handle event solicitation, the following function may be used

```
void fl_activate_event_callbacks(Window win);
```

A.6 Signals

The application program can elect to handle the receipt of a signal by registering a callback function that gets called when the signal is raised and caught

```
typedef void (*FL_SIGNAL_HANDLER)(int, void *);
void fl_add_signal_callback(int signal, FL_SIGNAL_HANDLER sh, void *data);
```

Only one callback per signal is permitted.

By default, `fl_add_signal_callback()` will store the callback function and initiate mechanism for the OS to deliver the signal when it occurs. When the signal is received by the library, the main loop will invoke the registered callback function when it is appropriate to do so. The callback function can make use of all **XForms**'s functions as well as Xlib functions as if there were reentrant. Further, a signal callback so registered is persistent and will cease to function only when explicitly removed.

It is very simple to use this routine. For example, to prevent a program from exiting prematurely due to interrupts, code fragment similar to the following can be used:

```
void clean_up(int signum, void *data)
{
     /* clean up, of course */
}

/* and somewhere after fl_initialize */

fl_add_signal_callback(SIGINT, clean_up, &mydata);
```

After this, whenever interrupt is detected, `clean_up` is called.

To remove a signal callback, the following routine should be used

```
void  fl_remove_signal_callback(int signal);
```

There are limitations with the default behavior outlined above. For example, on some platforms, there is no blocking of signals of any kind while handling a signal. In addition, use of `fl_add_signal_callback()` prevents the application program from using any, potentially more flexible, system signal handling routines on some platforms.

In light of these limitations, provisions are made so an application program may choose to take over the initial signal handling setup and receipt via various system dependent methods (*sigaction*(2) for example).

To change the default behavior of built-in signal facilities, the following routine should be called prior to any use of `fl_add_signal_callback()` with a true `flag`:

```
void fl_app_signal_direct(int flag)
```

After this call, `fl_add_signal_callback()` will not initiate any actions to receive a signal. The application program should handle the receipt and blocking of a signal (via, *signal*(2), *sigaction*(2), *sigprocmask*(2) etc.) When the signal is received by the application program, it should call the following routine to inform the main loop of the delivery of the signal `signum`

```
void fl_signal_caught(int signum);
```

This routine is the only one in the library that can be safely called from within a direct application signal handler. If multiple invocation of `fl_signal_caught()` occurs before the main loop is able to call the registered callback, the callback is called only once.

A.7 Idle callbacks and timeouts

For application programs that need to perform some light, but semi-continuous or periodic tasks, idle callback and timeouts (also `FL_TIMER+`) can be utilized.

To register an idle callback with the system, use the following routine

```
typedef int (*FL_APPEVENT_CB)(XEvent *, void *);
FL_APPEVENT_CB
fl_set_idle_callback(FL_APPEVENT_CB callback, void *user_data)
```

where `callback` is the function that will get called whenever the main loop is idle.

The time interval between any two consecutive invocations of the idle callback can vary considerably depending on the interface activity and other factors. A range between 50 and 300 milli-second should be expected.

It is possible to change the the condition (intervals of inactivity) based on which the main loop determines the idleness of the interface. To this end, the following is available

```
void fl_set_idle_delta(long msec)
```

where `msec` is the minimum interval of inactivity to be considered idle. However, it should be noted that under some conditions, an idle callback can be called sooner than the minimum interval.

If the timing of the idle callback is of concern, timeouts should be used. Timeouts are similar to idle callbacks but with the property that the user can specify a minimum time interval that must elapse before the callback is called. To register a timeout callback, the following routine can be used

```
typedef void (*FL_TIMEOUT_CALLBACK)(int, void *)
int fl_add_timeout(long msec,
                   FL_TIMEOUT_CALLBACK callback, void *data)
```

The function returns the timeout ID. When the time interval specified by `msec` (in milli-second) is elapsed, the timeout is removed, then the callback function is called. Although timeout offers some control over the timing, due to performance and cpu load compromises, the resolution at best is only 0.05 seconds, and can occasionally be in the 0.05-0.15 seconds range.

To remove a timeout before it triggers, use the following routine

```
void fl_remove_timeout(int ID)
```

where `ID` is the timeout ID returned by `fl_add_timeout()`.

See also Section 21.1 for the usage of `FL_TIMER` object.

Appendix B

Some Useful Functions

B.1 Misc. Functions

The following routine can be used to sound the keyboard bell (if capable),

```
void fl_ringbell(int percent)
```

where `percent` can range from -100 to 100 with 0 being the default volume setting of the keyboard. A value of 100 indicates maximum volume and a value of -100 indicates minimum volume (off). Note that not all keyboards support volume variations.

To get the user name who is running the application, you can use the following routine

```
const char *fl_whoami(void)
```

To get a string form of the current date and time, the following routine is available:

```
const char *fl_now(void)
```

The format of the string is of the form `Wed Jun 30 21:49:08 1993`.

The following time related routine might come in handy

```
void fl_gettime(unsigned long *sec, unsigned long *usec)
```

Upon function return, `sec` and `usec` are set to the current time, expressed in seconds and microseconds since 00:00 GMT January, 1970. This function is most useful for computing time differences.

B.2 Windowing Support

Some of these routines are used internally by the **Forms Library** as an attempt to localize window system dependencies and may be of some general use. Be warned that these routines may subject to changes, both in their API and/or functionality.

You can create and show a window with the following routines

```
Window fl_wincreate(const char *name)
Window fl_winshow(Window win);
```

where parameter `win` in `fl_winshow()` is the window ID returned by `fl_wincreate`. Between the creation and showing of the window, other attributes of the window can be set. Note a window so opened is always a top level window and uses all the **Forms Library**'f defaults (visual, depth etc.). Another thing about `fl_winshow()` is that it will wait and gobble up the first `Expose` event, and you can drawing into the window immediately after the function returns.

It is sometimes more convenient to create and show a window directly in one call

```
Window fl_winopen(const char *name)
```

This will open a (top-level) window with a title `name`. A window so opened can be drawn into as soon as the function returns, i.e., `fl_winopen()` waits until the window is ready to be drawn into.

The newly opened window will have the following default attributes

`event_mask` `ExposureMask`, `KeyPressMask`, `KeyReleaseMask`, `ButtonPressMask`, `ButtonReleaseMask`, `OwnerGrabButtonMask`, `ButtonMotionMask`, `PointerMotionMask`, `PointerMotionHintMask`, `StructureNotifyMask`

`backing_store` `fl_cntl.backingStore`

`class` `InputOutput`

`visual` same as **Forms Library**'s default.

`colormap` same as **Forms Library**'s default.

To make a top-level window a sub-window of another window, use the following routine

```
int fl_winreparent(Window win, Window new_parent)
```

The origin of the window `win` will be at the origin of the parent. At the time of the function call, both the window and the parent window must be valid windows.

By default, the newly opened window will have a size of 320 by 200 pixels and has no other constraints. You can modify the default or constraints using the following routines prior to calling `fl_winopen()`:

```
void fl_initial_winsize(FL_Coord w, FL_Coord h)
void fl_winsize(FL_Coord w, FL_Coord h)
```

These two routines set the preferred window size. `w` and `h` are the width and height of the window in pixels. `fl_winsize()` in addition will make the window non-resizeable (You can still resize the window programmatically) by setting the minimum and maximum window size to the requested size via `WMHints`. The effect of a window having this property is that it can't be interactively resized (provided the window manager cooperates).

It is sometimes desirable to have a window that is resizeable but only within a useful range. To set such a constraint, use the following functions:

```
void fl_winminsize(Window window, FL_Coord minw, FL_Coord minh)
void fl_winmaxsize(Window window, FL_Coord maxw, FL_Coord maxh)
```

These two routines can also be used after a window becomes visible. For windows to be created/opened, use 0 for the `window` parameter. For example, if we want to open a window of 640 by 480 pixels, and have it remain resizeable but within a permitted range, code similar to the following can be used:

```
fl_initial_winsize(640,480);
fl_winminsize(0, 100,100);
fl_winmaxsize(0, 1024,768)
win = fl_winopen("MyWin");
```

In addition to window size preference, you can also set preferred position of a window to be opened:

```
void fl_winposition(FL_Coord x, FL_Coord y)
```

where `x` and `y` are the coordinates of the upper-left corner of the window relative to the root window.

Or you can set the geometry in one function call

```
void fl_initial_wingeometry(FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h)

void fl_wingeometry(FL_Coord x, FL_Coord y,
                    FL_Coord w, FL_Coord h)
```

Again, the `fl_wingeometry()` will deny later interactive resizing.

There are other routines that can be used to change other aspects of the window to be created

```
fl_winaspect(Window win, FL_Coord x, FL_Coord y)
```

This will set the aspect ratio of the window in later interactive resizing.

To change the window title (and its associated icon title), use the following routine

```
void fl_wintitle(Window win, const char *title)
```

To change the icon title only, use the following routine

```
void fl_winicontitle(Window win, const char *title)
```

To install an icon for the window, use the following routine

```
 void fl_winicon(Window win, Pixmap icon, Pixmap mask)
```

You can suppress the window manager's decoration or make a window a transient one by using the following routines prior to creating the window

```
void fl_noborder(void)
void fl_transient(void)
```

You can also set the background of the window to a certain color using the following call

```
void fl_winbackground(Window win, unsigned long pixel)
```

It is possible to set the change size of a window to some discrete steps:

```
void fl_winstepsize(Window win, int xunit, int yunit)
```

where `xunit` and `yunit` are the number of pixels of changes per unit in the x- and y- directions respectively. Changes to the window size will be multiples of these units. after this call. Note that this only applies to interactive resizing.

To change constraints (size and aspect ratio) on an active window, you can use the following routine

```
void fl_reset_winconstraints(Window win)
```

The following routines are available to get information about an active window `win`:

```
void fl_get_winsize(Window win, FL_Coord *w, FL_Coord *h)
void fl_get_winorigin(Window win, FL_Coord *x, FL_Coord *y)
void fl_get_wingeometry(Window win, FL_Coord *x, FL_Coord *y,
                        FL_Coord *w, FL_Coord *h)
```

All units are in pixels. Origin of a window is measured from the upper left corner of the root window.

To change the size of window programmatically, the following routine is available:

```
int fl_winresize(Window win, FL_Coord neww, FL_Coord newh)
```

In addition to resizing the window, this routine will keep the original constraints. For example, if a window was not permitted to be interactively resized, after the resize, it remains unresizeable. Resizing is done by keeping the origin constant relative to the root window.

To move a window without resizing it, use the following call

```
void fl_winmove(Window win, FL_Coord newx, FL_Coord newy)
```

To move and resize a window, use the following routine

```
void fl_winreshape(Window win, FL_Coord newx, FL_Coord newy,
                    FL_Coord neww, FL_Coord newh)
```

To make a window invisible, use the following routine

```
void fl_winhide(Window win)
```

A hidden window can be shown again later using `fl_winshow()`.

To hide and destroy a window, use the following calls

```
void fl_winclose(Window win)
```

There will be no events generated from `fl_winclose()`, i.e., the function waits and eats all events for window `win`. In addition, this routine also removes all callbacks associated with the closed window.

The following routine can be used to check if a window is valid or not

```
int fl_winisvalid(Window win)
```

Note that excessive use of this function may negatively impact performance.

Usually an X application should work with window managers and accepts the keyboard focus assignment. In some special situations, explicit override of the keyboard focus might be warranted. To this end, the following routine exists:

```
void fl_winfocus(Window win)
```

After this call, keyboard input is directed to window `win`.

B.3 Cursors

XForms provides a convenience function to change the cursor shapes

```
void fl_set_cursor(Window win, int name)
```

where `win` must be a valid window identifier and `name` is one of the symbolic cursor names (shapes) defined by standard X or the integer value returned by `fl_create_bitmap_cursor()` or one of the **Forms Library**'s pre-defined symbolic names.

The X standard symbolic cursor names (all starts with `XC_`) are defined in `<X11/cursorfont.h>` (you don't need to explicitly include this as `forms.h` already does this for you). For example, to set a watch-shaped cursor for form `form` (after the form is shown), the following call may be made

```
fl_set_cursor(form->window,  XC_watch);
```

The **Forms Library** defined a special symbolic constants, `FL_INVISIBLE_CURSOR` that can be used to hide the cursor for window `win`:

```
fl_set_cursor(win, FL_INVISIBLE_CURSOR);
```

Depending on the structure of the application program, an `XFlush(fl_get_display())` may be required following `fl_set_cursor()`.

To reset the cursor to the **XForms**'s default (an arrow pointing northwest), use the following routine

```
void fl_reset_cursor(Window win)
```

To change the color of a cursor, use the following routine

```
void fl_set_cursor_color(int name, FL_COLOR fg, FL_COLOR bg)
```

where `fg` and `bg` are the foreground and background color of the cursor respectively. If the cursor is being displayed, the color change is visible immediately.

It is possible to use cursors other than those defined by the standard cursor font by creating a bitmap cursor

```
int fl_create_bitmap_cursor(const char *source, const char *mask,
                            int w, int h, int hotx, int hoty)
```

where `source` and `mask` are two (x)bitmaps. The mask defines the shape of the cursor. The pixels set to 1 in the mask defines which source pixels are displayed. If `mask` is null, all bits in source are displayed. `hotx` and `hoty` are the hotspot of the cursor (relative to the source's origin). The function returns the cursor ID which can be used in `fl_set_cursor()` and `fl_set_cursor_color()` etc.

Finally, there is a routine to create animated cursors where several cursors are displayed one after another:

```
int fl_create_animated_cursor(int *cur_names, int interval)
```

The function returns the cursor name that can be shown later via `fl_set_cursor()`. In the function call, `cur_names` is an array of cursor names (either X standard cursor or cursor name returned by `fl_create_bitmap_cursor()`) terminated by -1. Parameter `interval` indicates the time each cursor is displayed before displaying the next in the array. An interval about 150 milli-second is a good value for typical uses. Note that there is currently a limit of 24 cursors per animination sequence.

Internally animated cursor works by utilizing the timeout callback. This means that if the application blocks (thus the main loop has no chance of servicing the timeouts), the animation will not happen.

See demo `cursor.c` for an example use of the cursor routines.

B.4 Clipboard

Clipboard in the **Forms Library** is implemented using the X selection mechanism, more specifically the `XA_PRIMARY` selection. X selection is a general and flexible way of sharing arbitrary data among applications on the same server (The applications are of course not necessarily running on the same machine). The basic (and over-simplified) concept of the X selection can be summarized as follows: the X Server is the central point of the selection mechanism and all applications run on the server communicate with other applications through the server. The X selection is asynchronous in nature. Every selection has an owner (an application represented by a window) and every application can become owner of the selection or lose the ownership.

The clipboard in **Forms Library** is a lot simpler than the full-fledged X selection mechanism. The simplicity is achieved by hiding and handling some of the details and events that are of no interests to the application program. In general terms, you can think of a clipboard as a read-write buffer shared by all applications running on the server. The major functionality you want with a clipboard is the ability to post data onto the clipboard and request the content of the clipboard.

To post data onto the clipboard, use the following routine

```
typedef int (*FL_LOSE_SELECTION_CB)(FL_OBJECT *ob, long type)
int fl_stuff_clipboard(FL_OBJECT *ob, long type,
                       const void *data, long size,
                       FL_LOSE_SELECTION_CB callback)
```

where `size` is the size, in bytes, of the content pointed to by `data`. If successful, the function returns a positive value and the `data` would've being copied onto the clipboard. The `callback` is the function that will be called when other application takes ownership of the clipboard. For textual content, typically the application that loses the clipboard should undo the visual cues about the selection. If no action is required when losing the ownership, a null callback can be passed. The `ob` is used to obtain the window (owner) of the selection. `type` is currently unused. At the moment, the return value of `lose_selection_callback()` is also unused. The data posted onto the clipboard is available for all applications that manipulate `XA_PRIMARY` to use, such as *xterm* etc.

To request the current clipboard content, use the following routine

```
typedef int (*FL_SELECTION_CB)(FL_OBJECT *ob, long type,
                               const void * data, long size);

int fl_request_clipboard(FL_OBJECT *ob, long type,
                         FL_SELECTION_CB callback)
```

where `callback` is the callback function that gets called when the clipboard content is obtained. The content data passed to the callback function should not be modified.

One thing to remember is that the operation of the clipboard is asynchronous. Requesting the content of the clipboard merely asks the owner of the content for it and you will not have the content immediately (unless the asking object happens to own the selection). **XForms**main event loop takes care of the communication between the requesting object and the owner of the clipboard, and breaks up and re-assemble the content if it exceeds the maximum protocol request size (which has a guaranteed minimum of 16k bytes, but typically is larger). If the content of the clipboard is successfully obtained, the main loop invokes the `lose_selection` of the prior owner and then requesting object's callback function `callback`.

The function returns a positive number if the requesting object owns the selection (thus the callback invoked before the function returns) and 0 otherwise.

If there is no selection, the selection callback is called with an empty buffer and the length of the buffer is set to zero and `fl_request_clipboard()` returns -1.

Appendix C

Resources for Forms Library

Managing resources is an important part of programming with X. Typical X programs use extensive resource database/management to customize their appearances. With the help of **Form Designer**, there is little or no need to specify any resources for the default appearance of an application written using the **Forms Library**. Because of this, complete resource support is somewhat a low-priority task and currently only minimal support is available. Nevertheless, more complete and useful resource management system specific to the **Forms Library** can be implemented using the services provided by the **XForms**.

C.1 Current Support

At the moment, all built-in **XForms** resources have a top level class name `XForm` and a resource name `xform`. Because of this incomplete specification, most of the current resources are "global", in the sense that they affect all form windows. Eventually all resources will be fully resolved, e.g., to specify attribute `foo` of form `formName`, the resource name can be `appName.formName.foo` instead of (the current incomplete) `appName.xform.foo`.

The argument `app_opt` in `fl_initialize()` is a table of structures listing your application's command line options. The structure is defined as follows

```
typedef struct
{
    char *option;
    char *specifier;
    XrmOptionKind argKind;
    void *value;
} XrmOptionDescList, FL_CMD_OPT;
```

See *XrmGetResource*(3X11) for details.

After the initialization routine is called, all command line arguments, both **XForms** built-in and application specific ones, are removed from `argc` and `argv` and parsed into a standard XResources database. To read your application specific options, follow `fl_initialize()` with the following routine

```
int fl_get_app_resources(FL_resource *resource, int nresources);
```

Here `resource` is a table containing application specific resources in the following format:

```
typedef struct
{
    char *res_name;    /* resource name without application name    */
    char *res_class;   /* resource class                            */
    FL_RTYPE type;     /* C type of the variable                    */
    void  *var         /* variable that will hold the value         */
    char *defval;      /* default value in string form              */
    int nbytes;        /* buffer size for string var.               */
} FL_RESOURCE;
```

and the resource type `FL_RTYPE type` is one of the following

```
FL_SHORT          for short   variable
FL_BOOL           for boolean variable (int)
FL_INT            for int     variable
FL_LONG           for long    variable
FL_FLOAT          for float   variable
FL_STRING         for char [] variable
```

Note that the variable for `FL_BOOL` must be of type `int`. It differs from `FL_INT` only in the way the resources are converted, not in the way their values are stored. A boolean variable is considered to be true (1) if any one of `True, true, Yes, yes, On, on,` or `1` is specified as its value. For string variables, the length for the destination buffer must be specified.

`fl_get_app_resources()` simply looks up all entries specified in `FL_resource` structure in all databases after prefixing the resource name with the application name, which can be the new name introduced by the `-name` option.

Summarized below are the currently recognized **Forms Library** built-in resources:

Resource Name	Class	Type	Default	values
rgamma	Gamma	float	1.0	
ggamma	Gamma	float	1.0	
bgamma	Gamma	float	1.0	
visual	Visual	string	best	
depth	Depth	int	best	
doubleBuffer	DoubleBuffer	bool	false	
privateColormap	PrivateColormap	bool	false	
standardColormap	StandardColormap	bool	false	
sharedColormap	SharedColormap	bool	false	
pupFontSize	PupFontSize	int	12pt	
buttonFontSize	FontSize	int	10pt	
sliderFontSize	FontSize	int	10pt	
inputFontSize	FontSize	int	10pt	
browserFontSize	FontSize	int	10pt	
menuFontSize	FontSize	int	10pt	
choiceFontSize	FontSize	int	10pt	
ulPropWidth	ULPropWidth	bool	true	
ulThickness	ULThickness	int	1	
scrollbarType	ScrollbarType	string	thin	normal, thin, plain, nice
coordUnit	CoordUnit	string	pixel	
borderWidth	BorderWidth	int	3	

Again, “best" means that the **Forms Library** by default selects a visual that has the most depth.

By default, resource files are read and merged in the order as suggested by X11 R5 as follows:

- `/usr/lib/X11/app-defaults/<AppClassName>`
- `$XAPPRLESDIR/<AppClassName>`
- `RESOURCE_MANAGER` property as set using *xrdb* if `RESOURCE_MANAGER` is empty, `~/.Xdefaults`
- `$XENVIRONMENT` if `$XENVIORONMENT` is empty, `~/.Xdefaults`-*hostname*
- command line options

All options set via resources may *not* be the final values used because resource settings are applied at the time object/form is created, thus any modifications after that override the resource settings. For example, `buttonLabelSize`, if set, is applied at the time the button is created (`fl_add_button()`). Thus altering the size after the button is created via `fl_set_object_lsize()` overrides whatever is set by the resource database.

To run your application in `PseudoColor` with a depth of 8 and a thick underline, specify the following resources

```
appname*visual:          PseudoColor
```

```
appname*depth:          8
appname*ulThickness:    2
```

Since resources based on a form by form basis are yet to be implemented, there is no point specifying anything more specific although `appname.XForm.depth` etc. would work correctly.

An example

Let us assume that you have an application named `myapp`, and it accepts the options `-foo` *level* and `-bar`, and a filename. The proper way to initialize the **Forms Library** is as follows

```
FL_CMD_OPT cmdopt[] =
{
   {"-foo", "*.foo", XrmoptionSepArg, 0},
   {"-bar", ".bar", XrmoptionNoArg, "True"},
};

int foolevel, ifbar;
int deftrue;    /* can only be set thru resources */

FL_resource res[] =
{
    {"foo", "FooCLASS", FL_INT, &foolevel, "0"},
    {"bar", "BarCLASS",FL_BOOL, &ifbar,"0"},
    {"deftrue", "Whatever",FL_BOOL, &deftrue,"1"}
};

int
main(int argc, char *argv[])
{
   fl_initialize(&argc, argv ,"MyappClass", cmdopt, 2);
   fl_get_app_resources(res, 3);
   if(argc == 1)   /* missing filename */
      fprintf(stderr,"Usage %s: [-foo level][-bar] filename\n","myapp");
   /* rest of the program */
}
```

After this, both `foolevel` and `ifbar` are set either through resource files or command line options with the command line options overriding those set in the resource file. In case neither the command line nor the resource file specified the options, the default value string is converted.

There is another routine, the resource routine of the lowest level in **XForms**, might be useful if a quick& dirty option needs to be read:

```
const char *fl_get_resource(const char *res_name, const char *res_class,
                       FL_RTYPE type, char *defval, void *val, int nbytes)
```