Chapter 9

Command line arguments

To start the **Form Designer** simply type

```
fdesign [-xformoptions] [-fdesignoptions] [files[.fd]]
```

An initial window will be created and mapped. Depending on the window manager, you may have the option to interactively select where to place the window if `-geometry` option is missing. Next the program places the control panel on the screen. You can move this panel, if required, to the place you want (you can also change the default placement of the control panel via resources).

`fdesign` accepts all of the **XForms** command line options as well as the following

`-geometry` *geom* This option specifies the initial placement and size of the working area.

`-convert` *fd-file-list* Normally `fdesign` does its work interactively. This option causes the `fdesign` simply read a list of `fdesign` output file (the `.fd` files) and emit the corresponding C-routines and header files.

`-version` Prints current version and quits.

`-help` Prints a brief help message on command line options.

`-altformat` Generates an alternative output format.

`-border` Forces decorations on several windows so that you can move them easily.

`-unit` $point|pixel|mm|cp|cmm$ Outputs object sizes in units other than pixels. `cp` and `cmm` stand for centi-point (1/100 of a point) and centi-mm (1/100 of a milli-meter). For typical displays, $pixel$ and mm are too coarse and subject to round-off errors.

`-nocode` Suppresses the output of UI code. This can be handy if the UI code is not generated interactively, but rather generated by the *make* process using `fdesign -convert`.

`-I header` Changes the output include file from `forms.h` to `header`. Useful on systems where `forms.h` is renamed to something else or you need application specific constants/defines for the UI to function. In the later case, `header` may simply contain

```
#include "forms.h"
#define mystuff 1
```

`-main` Emits a main program with callback stubs. Can be useful for simple programs.

`-callback` Emits callback function template in a separate file.

`-lax` Suppresses syntax checking on variable and callback function names.

`-bw` *borderwidth* Changes default border width of the forms created.

Note that `-help, -version` and `-convert` do not require a connection to an X server.

If an output unit other than the default (`pixel`) is selected, all object sizes in the output file will be in the unit requested. This kind of UI has a fixed and device resolution independent size (in theory at least) and can be useful for drawing applications.

`fdesign` recognizes the following resources

workingArea.geometry	Geometry	string
control.border	XForm.Border	bool
control.geometry	Control.Geometry	string (position only)
attributes.geometry	Attributes.Geometry	string (position only)
attributes.background	Attributes.Background	string (e.g.,gray80)
align.geometry	Align.Geometry	string (position only)
help.geometry	Help.Geometry	string (position only)
convert	Convert	bool
unit	Unit	string
altformat	AltFormat	bool
xformHeader	XFormHeader	string
helpFontSize	HelpFontSize	int
main	Main	bool

Note that resource specification of `convert` requires an X connection.

In addition, all **XForms**'s resources specification can be used to influence the appearance of various panels. The most useful ones are the font sizes

*XForm.FontSize	all label font sizes
XForm.PupFontSize	all pup font sizes

Chapter 10

Creating Forms

10.1 Creating, changing and deleting forms

To create a new form press the button labeled New Form, indicate the enclosing box of the form and type in a (unique) name for the form. The form is shown in the main window and objects can be added to it.

There are two ways to change the size of a form at a later stage. The easiest way is to simply change the size of the main window and the form will resize itself to fit the new size. Or you can select the bottom box of the form, using the right mouse button. Next grab the box using the middle mouse at the lower-right corner and scale it. Note that objects lying outside the form will be invisible when the form is shown by the application program.

To change the name of the current visible form, press the button labeled Rename Form under the list of forms. You will be prompted for the new form name.

To delete a form, press the button labeled Delete Form. The current form will be removed.

10.2 Adding objects

To add an object, choose the class of the object from the object list in the middle of the control panel. Next drag the left mouse button on the form and an outline showing the current size of the object will appear. When the size is correct release the mouse button.

Note that the position and size of the object is rounded to multiples of 10 pixels. This can be changed. See below on alignments.

10.3 Selecting objects

To perform operations on objects that are already visible in the form, we first have to select them. Any mouse button can be used for selecting objects. Simply click it inside the object you want to select. A red outline will appear, indicating that the object is selected. In some cases when the currently selected object is large and encloses some smaller objects in it, left mouse might not be able to select the enclosed small objects. In this, use the the right mouse. Another way of selecting objects is to use the <TAB> key or the <F11> or the button labeled F11, which walks down the object list and selects an object upon each press.

It is also possible to select multiple objects. To this end, draw a box by dragging the mouse around all the objects you want to select. All objects that lie fully inside the box will be selected. Each selected object will get a red outline and a red bounding box is drawn around all of them.

To add objects to an already existing selection, hold down the <SHIFT> key and press the right mouse button inside the objects. You can remove objects from the selection by doing the same on an already selected object.

It is possible to select all objects (except for the backface) at once using the function key <F4>.

One note on the backface of the form. Although this is a normal object, it can not be treated in the same way as the other objects. It can be selected, but never in combination with other objects. Only two operations are allowed on it: changing its attributes and scaling it (which scales the size of the form).

10.4 Moving and scaling

Moving and scaling of objects is done using the middle mouse button. To move an object or a collection of objects to a new place, first select it (them) using the right mouse button as described above. Next press the middle mouse button inside the bounding box (not near one of the corners) and move the box to its new position.

To scale the object or objects, pick up the bounding box near one of its corners (inside the red squares) and scale it.

When holding the <SHIFT> key while moving an object or group of objects, first a copy of the object(s) is made and the copy is moved. This allows for a very fast way of duplicating (cloning) objects on the form: First put one on the form, change the attributes as required and next copy it.

For precise object movement, the cursor keys can be used. Each pressing of the four directional cursors moves the object 5 pixels. To change the step size, precedes the cursor keys with `0-9` with `0` indicating `10` pixels. If <SHIFT> is down, instead of moving the object, the object size is increased or decreased by the step size.

10.5 Aligning objects

Sometimes you have a number of objects and you want to align them in some way, e.g. centered or all starting at the same left position, etc. To this end press the button labeled Align. A special form will appear in the top right corner. You can leave this form visible as long as you want. You can hide it using the button Dismiss on the form or by clicking button Align again.

Now select the objects you want to align. Next, press one of the alignment buttons in the form. The buttons mean top row: flush left, horizontal center, flush right, horizontal equal distance (see below), bottom row: align bottoms, vertical center, align tops, vertical equal distance. Note that alignments are relative to the selection box, not to the form. Equal distance alignment means that between all the objects an equal sized gap is placed. The objects are kept in the same left to right or bottom to top order.

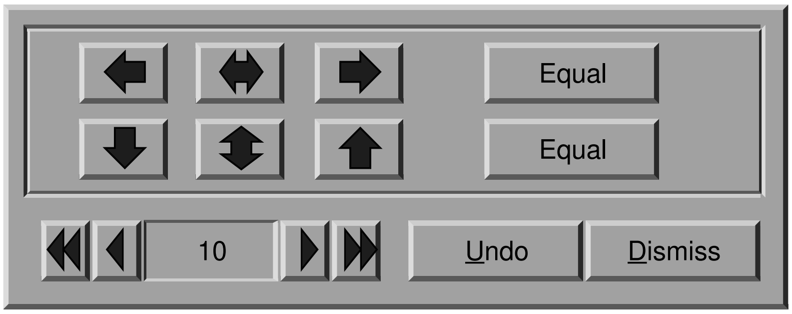

Figure 10.1: Object alignment control

In the alignment form you can also indicate the snapping size using the counter at the bottom. Choose 0 if you don't want to snap positions. Default snapping is 10 pixels. Snapping helps in making objects the same size and in making them nicely aligned.

The Undo undoes the last alignment change. It is an undo with a depth of 1, i.e., you can only undo the last change and an undo after an undo will undo itself. Note however, Any modification to the selected objects invalidates the undo buffer.

10.6 Raising and lowering

The objects in a form are drawn in the order in which they are added. Sometimes this is undesirable. For example, you might decide at a later stage to put a box around some buttons. Because you add this box later, it will be drawn over the buttons thus making the buttons invisible (if you put a framebox over a button, the button will be visible but appears to be inactive!). This is definitely not what you want. The **Form Designer** makes it possible to raise objects (bring them to the top) or lower them (put them at the bottom). So you can lower the box to move it under the buttons. Raising or lowering objects is very simple. First select the objects using the right mouse button and next press the function key <F2> to lower the selection or <F3> to raise it.

Another use of raising and lowering is to change the input field visitation order (via <TAB> key). Input fields focus order is the same as the order in which they are added to the form. This can become a problem if another input field is needed after the form is designed because this extra input field will always be the last among all input field on the form. Raising the objects becomes handy to solve this problem. What really happens when a object is raised is that the raised object becomes the last object added to the form. This means you can re-arrange the focus order by raising all input fields one by one in the exact order you want the focus order to be, and they will be added to the form in the order you raise them, thus the input focus order.

10.7 Setting attributes

To set attributes like type, color, label, etc., of an object first select it (using the right mouse button) and next press the function key <F1> (or click on the button labeled F1). If only one object is selected you can change all its attributes, including its label, name, etc. It is also possible to change the attributes of multiple objects as long as they all are of the same class. In this case you cannot change the label, name, etc. because you probably want them to remain different for the objects.

A form will appear in which you can indicate the different settings. Before we continue, the organization of the Attribute form and classification of attributes need a little explanation. Attributes of an object are divided into two categories. The Generic attributes are shared by all objects. These include type, color, label, callback functions etc. The Specific attributes are those that are specific to a particular object class, such as slider bounds, precision etc. When the Attribute form is first shown, only the Generic attributes are shown. Press on folder Spec to activate the object class specific attributes part (and press on button Generic to switch back to the generic attributes part).

10.8 Generic Attributes

10.8.1 Colors

Here you can indicate type, boxtype, and colors of the object, and style, size, alignment and color of the label. The type, boxtype, style, size and alignment are set using a choice object. To change it either use the left or middle mouse button to cycle through the possibilities, or use the right mouse button to get a menu with all choices. To change one of the colors, push the mouse on it. A box will appear showing the available colors in the internal color map. You can indicate the color you want with the mouse or use cancel to keep the color unchanged. (The color of the cancel button is the current color you are changing.) You can use the arrows to run through the color map to find other colors.

Once you are satisfied with the settings, press the button labeled Ready and the form will disappear. If you don't want to change the attributes after all press the button labeled Cancel.

10.8.2 Object names and call-back routines

Three more fields can be filled in in the attributes form: name, callback and argument. Name indicates the name of the object. If you type in a name here the object will be known to the application program under this name so that the program can refer to it. Take care that all object names used are different. They should be legal C variable names.[1] It is possible to use arrays of objects. E.g. if you define some objects as `obj[0]`, `obj[1]` and `obj[2]` the piece of C-code produced by the **Form Designer** will contain a declaration of an array `tt` of size 3. (Only one-dimensional arrays are treated correctly.)

Callback indicates the callback routine. If you type in something here, this routine will be bound to the object. In this case you also have to provide an argument that must be an integer (or cast

[1] Simple C++ variable names are also supported

to integer, as in (long)&variable). Of course, the application program will have to provide the callback routine.

Note that when copying objects these fields are also copied. This might lead to multiple objects with the same name. This will lead to undesired effects. So watch out for these after copying an object.

10.9 Object Specific Attributes

Currently not all objects can be initialized from with the **Form Designer**.

Depending on the objects, different attributes are shown that are considered to be intrinsic to the objects, such as slider bounds, precision etc. All the attributes should be self-explanatory and all changes made are shown immediately so you can see what effects the changes have on the object. Once satisfactory results are achieved, press button Accept to accept the settings (press on the folder Generic has the same effect). Two additional buttons Cancel and Restore are available to cancel the changes (and quit the attributes setting form) and restore the defaults, respectively.

One particular aspect of the pixmap/bitmap button initialization needs a little more explanation as the setting of button Use data has no effect on the appearance of the button but nonetheless affects the generated code. By default, button Use data is false, indicating the pixmap/bitmap file specified is to be loaded dynamically at run time via `fl_set_pixmapbutton_file()` (or the bitmap counterpart). If Use data is true, the specified file and its associated data will be `#include`'d at compile time so the data is part of the code. Depending on the application setup, you may choose one method over the other. In general, including the data in the code will make the code slightly larger, but it avoids the problems with not finding the specified file at runtime. The button Full Path only applies if Use Data is true. If Full Path is true, the pixmap file will be `#included` using the full path otherwise only the filename is used, presumably the compile process will take care of the path via `-I` flag in some system dependent way. In general, not using the full path is more flexible.

10.10 Cut, Copy and Paste

You can remove objects from the form by first selecting them and next pressing function key <F12> or double-clicking the left mouse button. The objects will disappear but are in fact saved in a buffer. You can put them back in the form, or in another form, by pasting them using <F10>. Note that only the last collection of deleted objects is saved in the buffer.

It is also possible to put a copy of the selection in the buffer using <F9>. This selection can now be put into the same form or into a different form. This allows for a simple mechanism of making multiple copies of a set of objects and for moving information from one form to another.

To clone the currently selected object, hold down the <SHIFT> key and drag the selected object. The cloned object will have exactly the same attributes as the original object except for object name and shortcut keys. Should these be cloned, the generated code would not be compilable (or cause runtime misbehavior).

10.11 Groups

As described in the tutorial about the **Forms Library**, sets of radio buttons must be placed inside groups. Groups are also useful for other purposes. E.g. you can hide a group inside an application program with one command. Hence, the **Form Designer** has some mechanism to deal with groups.

In the control panel there is a list of groups in the current form. As long as you don't have groups, this list will be empty. To create a group, select the objects that should come in the group and press the function key <F7>. You will be prompted for the name of the group. This should be a legal C variable name (under which the group will be known to the application program) or should be empty. This name will be added to the list. In this way you can create many groups. Note that each object can be in only one group. So if you select it again and put it in a new group, it will be removed from its old group. Groups that become empty this way automatically disappear from the list. (When putting objects in a group they will be raised. This is unavoidable due to the structure of groups.)

In the list of groups it is always indicated which groups are part of the current selection. (Only the groups that are fully contained in the selection are indicated, not those that are only partially contained in it.) It is also possible to add or delete groups in the current selection by pushing the mouse on their name in the list.

Note that there is no mechanism to add an object to a group directly. This can, however, be achieved using the following procedure. Select the group and the new object and press <F7> to group them. The old group will be discarded and a new group will be created. You only have to type in the group name again.

Sometimes you want to un-group the objects in an existing group, i.e., get them out of the group they are currently in. To this end simply select the group and press <F8>. (This only works if one group is selected.)

You can use the item *Rename group* under the Group menu to change the name of a selected group. If multiple groups are selected only the name of the first group is changed.

10.12 Hiding and showing

Sometimes it is useful to temporarily hide some objects in your form. In particular when you have sets of overlapping objects. To this end, select the objects you want to hide and press <F6>. The objects (though still selected) are now invisible. To show them again press <F5>. A problem might occur here. When you press <F5> only the selected objects will be shown again. But once an object is invisible it can no longer be selected. Fortunately, you can always use <F4> to select all objects, including the invisible ones, and press <F5> after that. It is better, though, to first group the objects before hiding them. Now you can select them by pressing the mouse on the group name in the group browser.

10.13 Testing forms

To test the current form, press the button labeled Test. The form will be displayed in the center of the screen. A panel will appear at the top right corner of the screen. This panel will show you the objects that will be returned and the callback routines called when working with the form. In this way you can verify whether the form behaves correctly and whether all objects have either callback routines or names (or both) associated with them. You can also resize the form (if the backface of the form allows resizing) to test the gravities. You can play with the form as long as you want. When ready, press the button Stop Testing.

Note that any changes you made, including the size of the form, to the form while testing do not show up when saving the form. E.g. filling in an input field or setting a slider does not mean that in the saved code the input field will be filled in or the slider set.

Chapter 11

Saving and loading forms

To save the set of forms created select the item *Save* or *Save As* from the File menu. You will be prompted for a file name using the file selector if the latter is selected. Choose a name that ends with `.fd`. e.g. `ttt.fd`.

The program will now generate three files `ttt.c`, `ttt.h` and `ttt.fd`. If these files already exist, backup copies of these are made (by appending `.bak` to the file names). `ttt.c` contains a piece of C-code that builds up the forms and `ttt.h` contains all the object and form names as indicated by the user. It also contains declaration of the defined callback routines.

Depending on the options selected from the Options menu, two more files may be emitted. Namely the main program and callback function templates. They are named `ttt_cb.c` and `ttt_main.c` respectively.

There are two different kind of formats for the C-code generated. The default format allows more than one instances of the form created and uses no global variables. The other format, activated by `altformat` on the command line, or from the Options menu by selecting *Alt Format*, uses global variables and does not allow more than one instantiation of the designed forms. However, this format has a global routine that creates all the forms defined, which by default is named `create_the_forms()` but it can be changed (see below).

Depending on which format is output, the application program typically only needs to include the header file and call the form creation routine.

To illustrate the differences between the two output formats and the typical way an application program is setup, we look at the following hypothetical situation: We have two forms, `foo` and `bar`, each of which contains several objects, say `fnobj1, fnobj2` etc. where `n=1,2`. The default output format will generate the following header file (`foobar.h`):

```
#ifndef FD_foobar_h_
#define FD_foobar_h_

/* call back routines if any */
extern void callback(FL_OBJECT *,long);

typedef struct
```

```
{
     FL_FORM *foo;
     void *vdata;
     char *cdata;
     long  ldata;
     FL_OBJECT *f1obj1;
     FL_OBJECT *f1obj2;
} FD_foo;

typedef struct
{
     FL_FORM *bar;
     void *vdata;
     cahr *cdata;
     long  ldata;
     FL_OBJECT *f2obj1;
     FL_OBJECT *f2obj2;
} FD_bar;

extern FD_foo *create_form_foo(void);
extern FD_bar *create_form_bar(void);

#endif /* FD_foobar_h */
```

and the corresponding C file:

```
#include "forms.h"
#include "foobar.h"

FD_foo *create_form_foo(void)
{
    FD_foo *fdui = (FD_foo *) fl_calloc(1, sizeof(FD_foo));

    fdui->foo = fl_bgn_form(....);
    fdui->f1obj1 = fl_add_xxxx(....);
    .....
    fl_end_form();

    fdui->foo->fdui = fdui;
    return fdui;
}

FD_bar *create_form_foo(void)
{
    FD_bar *fdui = (FD_bar *) fl_calloc(1, sizeof(FD_bar));

    fdui->bar = fl_bgn_form(....);
```

```
    fdui->f2obj1 = fl_add_xxxx(....);
    .....
    fl_end_form();

    fdui->bar->fdui = fdui;
    return fdui;
}
```

The application program would look something like the following:

```
#include "forms.h"
#include "foobar.h"

/* add call back routines here */

main(int argc, char *argv[])
{
    FD_foo *fd_foo;
    FD_bar *fd_bar;

    fl_initialize(...);
    fd_foo = create_form_foo();
    init_fd_foo(fd_foo);        /* application UI init routine */

    fd_bar = create_form_bar();
    init_fd_bar(fd_bar)         /* application UI init routine */

    fl_show_form(fd_foo->foo, ...);
    /* rest of the program */
}
```

As you see, `fdesign` generates a structure that groups together all objects on a particular form and the form itself into a structure for easy maintenance and access. The other benefit of doing this is that the application program can create more than one instances of the form if needed.

It is difficult to avoid globals in an event-driven callback scheme with the most difficulties occurring inside the callback function where another object on the same form may need to be accessed. Current setup makes it possible and relatively painless to achieve this.

There are a couple of ways to do this. The easiest and most robust way is to use the member `form->fdui`, which `fdesign` is set to pointing to the `FD_` structure in which the `form` is member. To illustrate how this is done, let's take the above two forms and try to access a different object from within a callback function.

```
fdfoo = create_form_foo();
...
```

and in the callback function of `ob` on form `foo`, you can access other objects as follows:

```
void callback(FL_OBJECT *ob, long data)
{
   FD_foo *fdfoo = ob->form->fdui;
   fl_set_object_xxx(fdfoo->f1obj2, ....);
}
```

Of course this setup still leaves the problems accessing objects on other forms unsolved although you can manually set the `form->u_vdata` to the other `FD_` structure: `fd_foo->form->u_vdata = fd_bar` or use the `vdata` field in the `FD_` structure itself: `fd_foo->vdata = fd_bar`.

The other method, not as easy as using `form->fdui` (because you get no help from `fdesign`), but just as workable, is simply use the `u_vdata` in `FD_` structure to hold the ID of the object that needs to be accessed. In case of a need to access multiple objects, there is a field `u_vdata` in both `FL_FORM` and `FL_OBJECT` structures you can use. You simply use the field to hold the `FD_` structure:

```
fdfoo = create_form_foo();
fdfoo->foo->u_vdata = fdfoo;
...
```

and in the callback function, you can access other objects as follows:

```
void callback(FL_OBJECT *ob, long data)
{
   FD_foo *fdfoo = ob->form->u_vdata;
   fl_set_object_xxx(fdfoo->f1obj2, ....);
}
```

Not pretty, but adequate for practical purposes. Note that the FD structure always has the form as the first entry and followed by `vdata`, `cdata` and `ldata`. Also a struct `FD_Any` is defined in the `forms.h`:

```
typedef struct
{
    FL_FORM *form;
    void *vdata;
    char *cdata;
    long  ldata;
} FD_Any;
```

you can use to cast a specific `FD_` structure get to the `vdata`.

Another alternative is to use the `FD_` structure created as the user data in the callback

```
fl_set_object_callback(obj, callback, (long)fdui);
```

and use the callback as follows[1]

```
void callback(FL_OBJECT *ob, long arg)
{
    FD_foo *fdfoo = (FD_foo *) arg;

    fl_set_object_lcol(fdfoo->f1obj1, FL_RED);
    ...
 }
```

Avoiding globals is, in general, a good idea, but as everything else, an excess of a good thing can be bad. Sometimes, simply making the `FD_` structure global makes a program clearer and more maintainable.

There still is another difficulty that might arise with the current setup. For example, in `f1obj1`'s callback we change the state of some other objects , say, `f1obj2` via `fl_set_button/input`. Now the state of `f1obj2` is changed and it needs to be handled. You probably don't want to put too much `f1obj2`'s handling code in `f1obj1`'s callback. In this situation, the following function comes in handy

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

`fl_call_object_callback(fdfoo->f1obj2)` will invoke the `f1obj2`'s callback in exactly the same way the main loop would and as far as `f1obj2` is concerned, it just handles the state change as if the user changed it.

The alternative format outputs something like the following:

```
/* callback routines */
extern void callback(FL_OBJECT *, long);

extern FL_FORM *foo, *bar;
extern FL_OBJECT *f1obj1, f1obj2 ...;
extern FL_OBJECT *f2obj1, f2obj2 ...;

extern void create_form_foo(void), create_form_bar(void);
extern void create_the_forms(void);
```

The C-routines:

```
FL_FORM *foo, *bar;
FL_OBJECT *f1obj1, *f1obj2 ...;
FL_OBJECT *f2obj1, *f2obj2 ...;
```

[1]This scheme is illegal as a pointer may be longer than a long, but in practice, it should work out ok on virtually all platforms.

```
void create_form_foo(void)
{
   if(foo)
     return;
   foo = fl_bgn_form(....);
   ...
}

void create_form_bar(void)
{
   if(bar)
     return;
   bar = fl_bgn_form(....);
   ...
}

void create_the_forms(void)
{
    create_form_foo();
    create_form_bar();
}
```

Normally the application program would look something like this:

```
#include "forms.h"
#include "foobar.h"

/* The call back routines */

main(int argc, char *argv[])
{
  fl_initialize(....);
  create_the_forms();
  /* rest of the program */
}
```

Note that although the C-routine file in both cases is easily readable, editing it is strongly discouraged. If you were to do so, you will have to redo the changes whenever you call `fdesign` again to modify the layout.

The third file created, `ttt.fd`, is in a format that can be read in by the **Form Designer**. It is easy readable ASCII but you had better not change it because not much error checking is done when reading it in. To load such a file select the *Open* from the File menu. You will be prompted for a file name using the file selector. Press your mouse on the file you want to load and press the button labeled Ready. The current set of forms will be discarded, and replaced by the new set. You can also merge the forms in a file with the current set. To this end select *Merge* from the File menu.

Chapter 12

Language Filters

This chapter discusses the language filter support in **Form Designer**, targeted primarily to the developers of other language bindings to **Forms Library**. As of this writing, the authors are aware of the following bindings

- `ada95` binding by G. Vincent Castellano (`gvc@ocsystems.com`),
- `perl` binding by Martin Bartlett (`martin@nitram.demon.co.uk`),
- `Fortran` binding by G. Groten (`zdv017@zam212.zam.kfa-juelich.de`) and Anke Haeming (`A.Haeming@kfa-juelich.de`), and
- `pascal` binding by Michael Van Canneyt (`michael@tfdec1.fys.kuleuven.ac.be`)
- `python` binding by Roberto Alsina (`ralsina@ultra7.unl.edu.ar`). It would appear that author of `python` binding is no longer working on it.

These bindings are of varying degree of beta-ness and support. It appears to the authors that the most convenient and flexible way of getting output in the targeted language is through external filters that are invoked transparently by the fdesign. This way, developers of the binding would have complete control over the translation of the default output from the fdesign to the target language and at the same time have the translation done transparently.

12.1 External filters

An external filter is a stand-alone program that works on the output of **Form Designer**, and translates the output to the target language. The filter can elect to work on the .fd or the c output or both simultaneously. However, in non-testing situations, the c output from **Form Designer** probably should be deleted by the filter once the translation is complete.

By default, **Form Designer** only outputs the .fd and c files. If the presence of `-ada`, `-perl`, `-python`, `-fortran` or `-pascal` command line options to **Form Designer**is detected, then after emitting the default output, **Form Designer** invokes the the external filter with the root filename (without the .fd extension) as an argument, together with possible other flags, to the filter. Any runtime error messages are presented to the user in a browser. The filter name by default is `fd2xxxx` where `xxxx` is the language name (such as `fd2python` etc.), which can be changed using the `-filter` command line option (or equivalent resources).

The resources that are relevant to the filter are listed below

Resource	Type	Default
language	string	C
filter	string	None

12.2 Command line arguments of the filter

Form Designer passes along the options that affect the output format to the filter. These options may or may not apply to the filter, most likely not if the filter works on the C file. For those that do not apply, the filter can simply ignore them, but shouldn't stop running because of these options.

`-callback` callback stubs are generated.

`-main` main stub is generated.

`-altformat` output in alternate format

`-compensate` emit size compensation code

Chapter 13

Generate hardcopies of the interface

A variety of tools are available that can be used to turn your carefully constructed (and hopefully pleasing) user interfaces into printed hardcopies or something appropriate for inclusion in your program document. Most of these involves saving a snapshot of your interface on the screen into a file. Then this file is translated into something that a printer can understand, such as PostScript. While this approach works, the resulting file is typically huge. Further, by taking a snapshot of the screen, the resolution of the output is limited by the screen resolution, which typically is much lower than the printer resolution. This is especially evident for text.

Another approach is to design the printing capabilities into the objects themselves so the GUI is somewhat output device independent in that it can render to different devices and X or the printer is just one of the devices. While this approach works better than screen snapshot, in general, it bloats the library unnecessarily. It is our observation that most of the time when a hardcopy of the interface is desired, it is for use in the application documentation. Thus we believe that there are ways to meet the needs of wanting hardcopies without bloating the library. Of course, some objects, such as xyplot, charts and possibly canvas (if vector graphics), that are dynamic in nature, probably should have some hardcopy output support in the library, even then, the relevant code should only be loaded when these specific support is actually used. This fattening problem is becoming less troublesome as computers are faster and typically with more RAMs on them nowadays.

fd2ps was designed to address the need of having a hardcopy of the interface for application documentation development. Basically, **fd2ps** is a translator that translates the **Form Designer** output directly into PostScript or Encapsulated PostScript in full vector graphics. The result is a small, may even be editable, PostScript file that you can print on a printer or include into other documents.

The translation can be done in two ways. One way is to simply give the **Form Designer** the command line option *-ps* to have it output POSTSCRIPT directly or you can run the **fd2ps** stand alone `fd2ps fdfile` where `fdfile` is the **Form Designer** output with or without the `.fd` extension. The output is written into a file named `fdfile.ps`.

The **fd2ps** accepts the following command line options when run stand alone

`-h` This option prints a brief help message.

`-p` This options requests Portrait output. By default, the orientation is switched to landscape automatically if the output would not fit on the page. This option overrides the default.

`-l` This option requests landscape orientation.

`-gray` This option requests all colors be converted to gray levels. By default, **fd2ps** outputs colors as specified in the `.fd` file.

`-bw` *n* This option specifies the object border width. By default, the border width specified in the `.fd` file is used.

`-dpi` *f* This option specifies the screen resolution on which the user interface was designed. You can use this flag to enlarge or shrink the designed size by giving a DPI value smaller or larger than the actual screen resolution. The default DPI is 85. If the `.fd` file is specified in device independent unit (point, mm etc), this flag has no effect. Also this flag does not change text size.

`-G` *f* This option specifies a value (gamma) that will be used to adjust the builtin colors. Larger the value, bright the colors. The default gamma is 1.

`-rgb` *file* The option specifies the path to the colorname database `rgb.txt` (It is used in parsing the colornames in XPM file). The default is `/usr/lib/X11/rgb.txt`. Environment variable `RGBFile` can be used to change this default.

`-pw` *f* This option changes the paper width used to center the GUI on a printed page. By default, the width is that of US Letter (i.e., 8.5 inches) unless the environment variable `PAPER` is defined.

`-ph` *f* This option changes the paper height used to center the output on the printed page. The default height is that of US Letter (i.e., 11 inches) unless the environment variable `PAPER` is defined.

`-paper` *name* This option specifies one of the standard paper names (thus setting the paper width and height simultaneously). The current understood papers are listed below.

Letter 8.5×11 inch.

Legal 8.5×14in.

A4 210×295mm.

B4 257×364mm.

B5 18×20cm.

B 11×17in.

Note 4×5in.

The **fd2ps** program understands the environment variable `PAPER`, which should be one of the above paper names.

Part III

An overview of all object classes

Chapter 14

Introduction

This part describes all different object classes that are available in the **Forms Library**.

All available object classes are summarized in Table 14.1.

For each class there is a section in this document that describes it. The section starts with a short description of the object, followed by the routine(s) to add it to a form. For (almost) all classes this routine has the same form

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

Here `type` is the type of the object in its class. Most classes have many different types. They are described in the section. x, y, w and h give the left bottom corner and the width and height of the bounding box of the object. `label` is the label that is placed inside or next to the object. For each object class the default placement of the label is described. When the label starts with the character `@` the label is not printed but replaced by a symbol instead.

For each object class there is also a routine

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

that only creates the object but does not put it in the form. This routine is useful for building hierarchical object classes. The routine is not described in the following sections.

An important aspect of objects is how interaction is performed with them. First of all there is the way in which the user interacts with the object and secondly it is indicated when the object changes status and is returned to the application program for some action. Both are described in the section.

Object attributes can be divided into generic and object specific ones. For generic attributes (e.g., the object label size), the routines that change them always start with `fl_set_object_xxx()` where `xxx` is the name of the attribute. When a specific object is created and added to a form, it inherits many aspects of the generic object or initializes the object attributes to its needed defaults.

Name	Description
Static Objects	
Box	Rectangular areas to visually group objects.
Frame	A box with an empty inside region.
Labelframe	A frame with label on the frame.
Text	Simple one line labels.
Bitmap	Displays an X11 bitmap.
Pixmap	Displays a pixmap using the XPM library.
Clock	A clock.
Chart	Bar-charts, pie-charts, strip-charts, etc.
Button Like Objects	
Button	Many different kinds and types of buttons that the user can push to indicate certain settings or actions.
Valuator Objects	
Slider	Both vertical and horizontal sliders to let the user indicate some float value.
Scrollbar	Sliders plus two directional buttons.
Dial	A dial to let the user indicate a float value.
Positioner	Lets the user indicate an (x, y) position with the mouse.
Counter	A different way to let a user step through values.
Input Objects	
Input	Lets the user type in an input string.
Choice Objects	
Menu	Both pop-up and drop-down menus can be created.
Choice	Can be used to let the user make a choice from a set of items.
Browser	A text browser with a slider. Can be used for making selections from sets of choices.
Container Objects	
Tabbed Folders	A tabbed folder is a compound object capable of holding multiple groups of objects.
Menu bar	A menubar is a collection of individual menus.
Other Objects	
Timer	A timer that runs from a set time towards 0. Can e.g. be used to do default actions after some time has elapsed.
XYPlot	XYPlot shows simple 2D xy-plot from a tabulated function or a datafile. Data points can be interactively manipulated and retrieved.
Pop-ups	Pop-ups are used by menu and choice. it can also be used stand-alone to allow the user to make selections among many choices.
Canvas	Canvases are managed plain X windows. It differs from a raw application window only in the way its geometry is managed, not in the way various interaction is set up.

Table 14.1: List of object classes

Thus, in the following sections, only the object specific routines are documented. Routines that set generic object attributes are documented in Part V.

When appropriate, the effect of certain (generic) attributes of the objects on the specific object is discussed. In particular it is indicated what the effect of the routine `fl_set_object_color()` is on the appearance of the object. Also some remarks on possible boxtypes are made.

Chapter 15

Static objects

15.1 Box

Short description

Boxes are simply used to give the dialogue forms a nicer appearance. They can be used to visually group other objects together. The bottom of each form is a box.

Adding an object

To add a box to a form you use the routine

```
FL_OBJECT *fl_add_box(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 default placed centered in the box.

Types

The following types are available:

`FL_UP_BOX`	A box that comes out of the screen.
`FL_DOWN_BOX`	A box that goes down into the screen.
`FL_FLAT_BOX`	A flat box without a border.
`FL_BORDER_BOX`	A flat box with a border.
`FL_FRAME_BOX`	A flat box with an engraved frame.
`FL_SHADOW_BOX`	A flat box with a shadow.
`FL_ROUNDED_BOX`	A rounded box.
`FL_RFLAT_BOX`	A rounded box without a border.
`FL_RSHADOW_BOX`	A rounded box with a shadow.
`FL_OVAL_BOX`	An elliptic box.
`FL_NO_BOX`	No box at all, only a centered label.

Interaction

No interaction takes place with boxes.

Other routines

No other routines are available for boxes.

Attributes

Color1 controls the color of the box.

Remarks

Do not use `FL_NO_BOX` type if the label is to change during the execution of the program.

Boxes are used in most demo's. Also see Fig. 3.1.

15.2 Frame

Short description

Frames are simply used to give the dialogue forms a nicer appearance. They can be used to visually group other objects together. Frames are almost the same as a box, except that the interior of the bounding box is not filled. Use of frames can speed up drawing in certain situations. For example, to place a group of radio buttons within an `FL_ENGRAVED_FRAME`. If we were to use an `FL_FRAME_BOX` to group the buttons, visually they would look the same. However, the latter is faster as we don't have to fill the interior of the bounding box and can also reduce flicker. Frames are useful in decorating free objects and canvases.

Adding an object

To add a frame to a form you use the routine

```
FL_OBJECT *fl_add_frame(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 except that the frame is drawn *outside* of the bounding box (so a flat box of the same size just fills the inside of the frame without any gaps). The label is by default placed centered inside the frame.

Types

The following types are available:

`FL_NO_FRAME`	Nothing is drawn
`FL_UP_FRAME`	A frame appears coming out of the screen
`FL_DOWN_FRAME`	A frame that goes down into the screen.
`FL_BORDER_FRAME`	A frame with a simple outline
`FL_ENGRAVED_FRAME`	A frame appears to be engraved.
`FL_EMBOSSED_FRAME`	A frame appears embossed .
`FL_ROUNDED_FRAME`	A rounded frame.
`FL_OVAL_FRAME`	An elliptic box.

Interaction

No interaction takes place with frames.

Other routines

None.

Attributes

Color1 controls the color of the frame if applicable. Boxtype attribute does not apply to the frame class.

Remarks

It may be faster to use frames instead of boxes for text that is truly static. See `freedraw.c` for an example use of frame objects.

15.3 LabelFrame

Short description

A label frame is almost the same as a frame except that the label is placed *on* the frame (See Fig. 15.1) instead of inside or outside of the bounding box as in a regular frame.

Adding an object

To add a labelframe to a form you use the routine

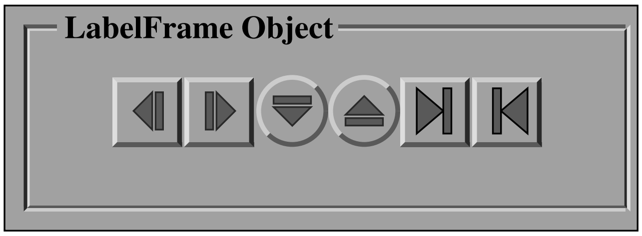

Figure 15.1: Labelframe Classes

```
FL_OBJECT *fl_add_labelframe(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 except that the frame is drawn *outside* of the bounding box (so a flat box of the same size just fills the inside of the frame without any gaps). The label is by default placed centered inside the frame.

Types

The following types are available:

FL_NO_FRAME	Nothing is drawn
FL_UP_FRAME	A frame appears coming out of the screen
FL_DOWN_FRAME	A frame that goes down into the screen.
FL_BORDER_FRAME	A frame with a simple outline
FL_ENGRAVED_FRAME	A frame appears to be engraved.
FL_EMBOSSED_FRAME	A frame appears embossed .
FL_ROUNDED_FRAME	A rounded frame.
FL_OVAL_FRAME	An elliptic box.

Interaction

No interaction takes place with frames.

Other routines

None.

Attributes

Color1 controls the color of the frame if applicable. Color2 controls the background color of the label. Boxtype attribute does not apply to the labelframe class.

Remarks

You can not draw a label inside or outside of the frame box. If you try, say, by requesting `FL_ALIGN_CENTER`, the label is drawn using `FL_ALIGN_TOP_LEFT`.

15.4 Text

Short description

Text objects simply consist of a label possibly placed in a box.

Adding an object

To add a text to a form you use the routine

```
FL_OBJECT *fl_add_text(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 by default placed flushed left in the bounding box.

Types

Only one type of text exists: `FL_NORMAL_TEXT`.

Interaction

No interaction takes place with text objects.

To set or change the text shown, use `fl_set_object_label()`

Other routines

No other routines are available for texts.

Attributes

Any boxtype can be used for text. Color1 controls the color of the box. The color of the text is controlled by lcol as usual. However, if the text is to change dynamically, `NO_BOX` should not be used.

Remarks

Don't use boxtype `FL_NO_BOX` if the label is to change dynamically.

Note that there is almost no difference between a box with a label and a text. The only difference lies in the position where the text is placed and the fact that text is clipped to the bounding box. Text is normally placed inside the box at the left side. This helps you putting different lines of text below each other. Labels inside boxes are default centered in the box. You can change the position of the text inside the box using the routine `fl_set_object_lalign()`. In contrast to boxes, different alignments for text always place the text inside the box rather than outside the box.

15.5 Bitmap

Short description

A bitmap is a simple bitmap shown on a form.

Adding an object

To add a bitmap to a form you use the routine

```
FL_OBJECT *fl_add_bitmap(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 by default placed below the bitmap. The bitmap will be empty.

Types

Only the type `FL_NORMAL_BITMAP` is available.

Interaction

No interaction takes place with a bitmap. For bitmap that interacts, see Section 16.1 on `bitmapbutton`. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static bitmap.)

Other routines

To set the actual bitmap being displayed use

```
void fl_set_bitmap_data(FL_OBJECT *ob, int w, int h, unsigned char *bits)
```

```
void fl_set_bitmap_file(FL_OBJECT *ob, const char *file);
```

bits contains the bitmap data as a character string. file is the name of the file that contains bitmap data. A number of bitmaps can be found in /usr/include/X11/bitmaps or similar places. The X program bitmap can be used to create bitmaps.

Two additional routines are provided to make a Pixmap from a bitmap file or data

```
Pixmap fl_read_bitmapfile(Window win, const char *filename,
                          unsigned *width, unsigned *height,
                          int *hotx, int *hoty)

Pixmap fl_create_from_bitmapdata(Window win, const char *data,
                                 int width, int height)
```

where win is any window ID in your application and other parameters have the obvious meanings. If there is no window created yet, fl_default_win() may be used.

Note pixmaps created by the above routines have a depth of 1 and should be displayed using XCopyPlane.

Attributes

Label color controls the foreground color of the bitmap. Color1 controls the background color of the bitmap (and the color of the box). Color2 is not used.

Remarks

See demo33.c for a demo of a bitmap.

15.6 Pixmap

Short description

A pixmap is a simple pixmap (color icons) shown on a form.

Adding an object

To add a bitmap to a form you use the routine

```
FL_OBJECT *fl_add_pixmap(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 by default placed below the pixmap. The pixmap will be empty.

Types

Only the type FL_NORMAL_PIXMAP is available.

Interaction

No interaction takes place with a pixmap. For pixmap that interacts, see Section 16.1 on pixmapbutton. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static pixmap.)

Other routines

A pixmap file (usually with suffix xpm) is an ASCII file that contains the definition of the pixmap as a char pointer array that can be included directly into a C (or C++) source file.

To set the actual pixmap being displayed, use one of the following routines:

```
void fl_set_pixmap_file(FL_OBJECT *ob, const char *file);

void fl_set_pixmap_data(FL_OBJECT *ob, char **data)
```

In the first routine, you specify the pixmap by the filename that contains it. In the second routine, you #include the pixmap at compile time and use the pixmap data (an array of char pointers) directly. Note that both of these functions do *not* free the old pixmaps associated with the object. If you're writing a pixmap browser type applications, be sure to free the old pixmaps using fl_free_pixmap_pixmap prior to calling these two routines.

To obtain the pixmap ID currently being displayed, the following routine can be used

```
Pixmap fl_get_pixmap_pixmap(FL_OBJECT *ob, Pixmap *id, Pixmap *mask);
```

In some situations, you might already have a Pixmap resource ID, e.g., from fl_read_pixmapfile(), you can use the following routine to change the the pixmap

```
void fl_set_pixmap_pixmap(FL_OBJECT *ob, Pixmap id, Pixmap mask)
```

where mask is used for transparency (See fl_read_pixmapfile().) Use 0 for mask if no special clipping attributes are desired.

This routine does not free the pixmap ID nor the mask already associated with the object. Thus if you no longer need the old pixmaps, they should be freed prior to changing the pixmaps using the following routine

```
void fl_free_pixmap_pixmap(FL_OBJECT *ob);
```

This routine in addition to freeing the pixmap and the mask, it also frees the colors the pixmap allocated.

Pixmaps are by default displayed centered inside the bounding box. However, this can be changed using the following routine

```
void fl_set_pixmap_align(FL_OBJECT *ob, int align, int dx, int dy)
```

where `align` is the same as that used for labels. See Section 3.11.3 for a list. `dx` and `dy` are extra margins to leave in addition to the object border width. By default, `dx` and `dy` are set to 3. Note that although you can place a pixmap outside of the bounding box, it probably is not a good idea.

Attributes

By default, if a pixmap has more colors than that available in the colormap, the library will use substitute colors that are judged "close enough". This closeness is defined as the difference between the requested color and color found being smaller than some pre-set threshold values between 0 and 65535 (0 means exact match). The default thresholds are 40000 for red, 30000 for green and 50000 for blue. The change these defaults, use the following routine

```
void fl_set_pixmap_colorcloseness(int red, int green, int blue);
```

Remarks

The following routines may come in handy to read a pixmap file into a Pixmap

```
Pixmap fl_read_pixmapfile(Window win, const char *filename,
                   unsigned *width, unsigned *height,
                   Pixmap *shape_mask, int *hotx, int *hoty,
                   FL_COLOR tran)
```

where `win` is the window in which the pixmap is to be displayed. If the window is yet to be created, you can use the default window `fl_default_window()`. Parameter `shape_mask` is set to a Pixmap, if not null, that can be used as a clip mask to achieve transparency. `hotx` and `hoty` are the center of the pixmap (useful if the pixmap is to be used as a cursor). Parameter `tran` is currently un-used.

If you have already had the pixmap data in memory, the following routine may be used

```
Pixmap fl_create_from_pixmapdata(Window win, char **data,
                          unsigned *width, unsigned *height,
                          Pixmap *shape_mask,
                          int *hotx, int *hoty, FL_COLOR tran)
```

All parameters have the same meaning as in `fl_read_pixmapfile`.

Note the **Forms Library** handles transparency, if specified in the pixmap file or data, for pixmap and pixmapbutton objects. However, when using `fl_read_pixmapfile` or `fl_create_from_pixmapdata`, the application programmer is responsible to set the clip mask in appropriate GCs.

Finally there is a routine that can be used to free a Pixmap

```
void fl_free_pixmap(Pixmap Id)
```

You will need the XPM library (version 3.4c or later) developed by Arnaud Le Hors and GROUPE BULL (`lehors@sophia.inria.fr`) to use pixmap. XPM library can be obtained from many X distribution/mirror sites via anonymous ftp or web (`ftp://ftp.x.org/contrib` and `ftp://avahi.inria.fr/pub/xpm` are the official site for anonymous ftp and `http://www.inria.fr/koala/lehors/xpm.html` is the home page).

15.7 Clock

Short description

A clock object simply displays a clock on the form.

Adding an object

To add a clock to a form you use the routine

```
FL_OBJECT *fl_add_clock(int type, FL_Coord x, FL_Coord y,
                        FL_Coord w, FL_Coord h, char label[])
```

The meaning of the parameters is as usual. The label is by default placed below the clock.

Types

The following types are available:

`FL_ANALOG_CLOCK`	An analog clock complete with the second hand.
`FL_DIGITAL_CLOCK`	A digital clock.

Interaction

No interaction takes place with clocks.

Other routines

To get the displayed time (local time as modified by the adjustment described below) use the following routine

```
void fl_get_clock(FL_OBJECT *obj, int *h, int *m, int *s)
```

Upon function return, the parameters are set as follows: `h` is between 0–23 indicating the hour, `m` is between 0–59 indicating the minutes and `s` is between 0–59 indicating the seconds.

To display time other than the local time, use the following routine

```
long fl_set_clock_adjustment(FL_OBJECT *ob, long adj)
```

where `adj` is in seconds. For example, to display a time that is one hour behind the local time, an adjustment of -3600 can be used. The function returns the old adjustment value.

By default, the digital clock uses 24hr system. You can switch the display to 12hr system (am-pm) by using the following routine

```
void fl_set_clock_ampm(FL_OBJECT *ob, int yes_no)
```

Attributes

Never use `FL_NO_BOX` as boxtype for a digital clock.

Color1 controls the color of the background, color2 the color of the hands.

Remarks

See `flclock.c` for an example of the use of clocks.

See also Page 275 for other time related routines.

15.8 Chart

Short description

The chart object gives you an easy way to display a number of different types of charts like bar-charts, pie-charts, line-charts, etc. They can either be used to display some fixed chart or a changing chart (e.g. a strip-chart). Values in the chart can be changed and new values can be added which makes the chart move to the left, i.e., new entries appear at the right and old entries disappear at the left. This can be used to e.g. monitor some process.

Adding an object

To add a chart object to a form use the routine

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

It shows an empty box on the screen with the label below it.

Types

The following types are available:

FL_BAR_CHART	A bar-chart
FL_HORBAR_CHART	A horizontal bar-chart
FL_LINE_CHART	A line-chart
FL_FILLED_CHART	A line-chart but area below curve is filled
FL_SPIKE_CHART	A chart with a vertical spike for each value
FL_PIE_CHART	A pie-chart
FL_SPECIALPIE_CHART	A pie-chart with displaced first item

All charts except pie-charts can display positive and negative data. Pie-charts will ignore values that are ≤ 0. The maximal number of values displayed in the chart can be set using the routine `fl_set_chart_maxnumb()`. The number must be bounded by `FL_CHART_MAX` which is 512. Switching between different types can be done without any complications.

Interaction

No interaction takes place with charts.

Other routines

There are a number of routines to change the values in the chart and to change its behavior. To clear a chart use the routine

```
void fl_clear_chart(FL_OBJECT *obj)
```

To add an item to a chart use

```
void fl_add_chart_value(FL_OBJECT *obj, double val,
                        const char *text, int col)
```

Here val is the value of the item, text is the label to be associated with the item (can be empty) and col is an index in the colormap (FL_RED etc) that is the color of this item. The chart will be redrawn each time you add an item. This might not be appropriate if you are filling a chart with values. In this case put the calls between fl_freeze_form() and fl_unfreeze_form().

By default, the label is drawn with tiny font in black. You can change the font style, size or color using the following routine

```
void fl_set_chart_lstyle(FL_OBJECT *ob, int fontstyle)

void fl_set_chart_lsize(FL_OBJECT *ob, int fontsize)

void fl_set_chart_lcolor(FL_OBJECT *ob, int color)
```

Note that fl_set_chart_lcolor() only affects the label color of subsequent items, not the items already created.

You can also insert a new value at a particular place using

```
void fl_insert_chart_value(FL_OBJECT *obj, int index,
                           double val, const char *text, int col)
```

index is the index before which the new item should be inserted. The first item is number 1. So, for example, to make a strip-chart where the new value appears at the left, each time insert the new value before index 1.

To replace the value of a particular item use the routine

```
void fl_replace_chart_value(FL_OBJECT *obj, int index,
                            double val, const char *text, int col)
```

Here index is the index of the value to be replaced. The first value has an index of 1, etc.

Normally, bar-charts and line-charts are automatically scaled in the vertical direction such that all values can be displayed. This is often not wanted when new values are added from time to time. To set the minimal and maximal value displayed use the routine

```
void fl_set_chart_bounds(FL_OBJECT *obj, double min, double max)
```

To return to automatic scaling choose min = max = 0.0.

Also the width of the bars and distance between the points in a line-chart are normally scaled. To change this use

```
void fl_set_chart_autosize(FL_OBJECT *obj, int autosize)
```

with autosize = 0. In this case the width of the bars will be such that the maximal number of items fits in the box. This maximal number (default FL_CHART_MAX) can be changed using

```
void fl_set_chart_maxnumb(FL_OBJECT *obj, int maxnumb)
```

where maxnumb is the maximal number of items to be displayed.

Attributes

Don't use `FL_NO_BOX` for a chart object if it changes value. Color1 controls the color of the box.

Remarks

See `chartall.c` and `chartstrip.c` for examples of the use of chart objects.

Chapter 16

Button like objects

16.1 Button

Short description

A very important class of objects are the buttons. Buttons are placed on the form such that the user can push them with the mouse. Different types of buttons exist: buttons that return to their normal position when the user releases the mouse, buttons that stay pushed until the user pushes them again and radio buttons that make other buttons be released.

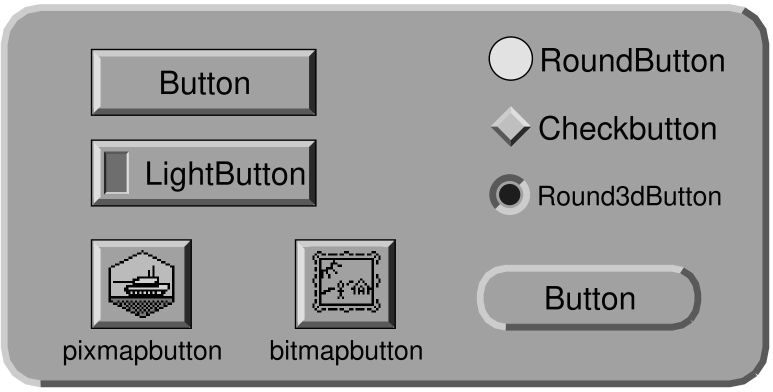

Figure 16.1: Button Classes

Also different shapes of buttons exist. Normal buttons are rectangles that come out of the background. When the user pushes them they go into the background (and possibly change color). Lightbuttons have a small light inside them. Pushing the button switches the light on. Round buttons are simple circles. When pushed, a colored circle appears inside them. Bitmap and pixmap buttons are buttons whose labels are graphics rather than text.

Adding an object

To add buttons use one of the following routines:

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

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

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

FL_OBJECT *fl_add_round3dbutton(int type, FL_Coord x, FL_Coord y,
                              FL_Coord w,FL_Coord h, const char *label)

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

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

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

FL_OBJECT *fl_add_scrollbutton(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 by default placed inside the button for button and lightbutton. For roundbutton, round3dbutton, bitmapbutton and pixmapbutton, it is placed to the right of the circle and to the bottom of the bitmap/pixmap respectively. For scrollbutton, the label must be of some pre-determined string that indicates the direction of the scroll arrow.

Types

The following types of buttons are available:

FL_NORMAL_BUTTON	Returns value when released.
FL_PUSH_BUTTON	Stays pushed until user pushes it again.
FL_MENU_BUTTON	Returns value when pushed.
FL_TOUCH_BUTTON	Returns value as long as the user pushes it.
FL_RADIO_BUTTON	Push button that switches off other radio buttons.
FL_HIDDEN_BUTTON	Invisible normal button.
FL_INOUT_BUTTON	Returns value both when pushed and when released.
FL_RETURN_BUTTON	Like a normal button but reacts on the <Return> key.
FL_HIDDEN_RET_BUTTON	Invisible return button.

Except for the FL_HIDDEN_BUTTON and FL_HIDDEN_RET_BUTTON, which are invisible, they all look similar on the screen but their function is quite different. Each of these buttons gets pushed down when the user presses the mouse on top of it. What actually happens when the user does so depends on the type of the button. An FL_NORMAL_BUTTON, FL_TOUCH_BUTTON and FL_INOUT_BUTTON gets released when the user releases the mouse button. Their difference lies in the moment at which the interaction routines return them (see below). A FL_PUSH_BUTTON remains pushed and is only released when the user pushes it again. A FL_RADIO_BUTTON is a push button with the following extra property. Whenever the user pushes a radio button, all other pushed radio buttons in the form (or in a group) are released. In this way the user can make its choice among some possibilities. A FL_RETURN_BUTTON behaves like a normal button, but it also reacts when the <Return> key on the keyboard is pressed. When a form contains such a button (of course there can only be one) the <Return> key can no longer be used to move between input fields. For this the <Tab> key must be used.

A FL_HIDDEN_BUTTON behaves like a normal button but is invisible. A FL_HIDDEN_RET_BUTTON is like a hidden button but also reacts to <Return> key presses.

Interaction

FL_NORMAL_BUTTONs, FL_PUSH_BUTTONs, FL_RADIO_BUTTONs, FL_RETURN_BUTTONs and FL_HIDDEN_BUTTONs are returned at the moment the user releases the mouse after having pressed it on the button. An FL_INOUT_BUTTON is returned both when the user presses it and when the user releases it. A FL_TOUCH_BUTTON is returned all the time as long as the user keeps it pressed. A FL_RETURN_BUTTON and a FL_HIDDEN_RET_BUTTON are also returned when the user presses the <Return> key.

See demo butttypes.c for a feel of the different button types.

Other routines

The application program can also set a button to be pushed or not itself without a user action. To this end use the routine

```
void fl_set_button(FL_OBJECT *obj, int pushed)
```

pushed indicates whether the button should be pushed (1) or released (0). When setting a FL_RADIO_BUTTON to be pushed this automatically releases the currently pushed button if different. Also note that this routine only simulates the visual appearance and perhaps some internal states, it does *not* affect the program flow in anyway, i.e., setting a button being pushed does not invoke its callback or results in the button returned to the program. For that, fl_trigger_object() is needed or more conveniently follow fl_set_button() with fl_call_object_callback().

To figure out whether a button is pushed[1] or not use

```
int fl_get_button(FL_OBJECT *obj)
```

[1] fl_mouse_button() can also be used

Sometimes you want to give the button a different meaning depending on which mouse button pressed it. To find out which mouse button was used at the last push (or release) use the routine

```
int fl_get_button_numb(FL_OBJECT *obj)
```

It returns one of the constants FL_LEFT_MOUSE, FL_MIDDLE_MOUSE and FL_RIGHT_MOUSE indicating the physical location of the mouse button on the mouse. If the last push is triggered by a shortcut (see below), the function returns the keysym (ascii value if ASCII) of the key plus FL_SHORTCUT. For example, if a button has a shortcut <CNTRL> C, the button number returned upon activation of the shortcut would be FL_SHORTCUT 3+.

If more information is desired about the last event, use

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

In a number of situations it is useful to define a keyboard equivalent to a button. E.g., you might want to define that ^Q (<CNTRL> Q) has the same meaning as pressing the Quit button. This can be achieved using the following call:

```
void fl_set_button_shortcut(FL_OBJECT *obj, const char *str,
                            int showUL)
```

Note that str is a string, not a character. This string should contain all the characters that correspond to this button. So, e.g., if you use string "^QQq" the button will react on the keys q, Q and <CNTRL> Q. (As you see you should use the symbol ^ to indicate the control key. Similarly you can use the symbol # to indicate the <ALT> key.) Be careful with your choices. When the form also contains input fields you probably don't want to use the normal printable characters because they can no longer be used for input in the input fields. Shortcuts always go before input fields. Other special keys, such as <F1> etc., can also be used as shortcuts. See Section 24.1 for details. Finally realize that a return button is in fact a normal button with the <Return> key as a shortcut. So don't change the shortcuts for such a button.

If the second parameter showUL is true, and one of the letters in the object label matches the shortcut, the matching letter will be underlined. This applies to non-printable characters (such as #A) as well in the sense that if the label contains letter a or A, it will be underlined (i.e., special characters such as # and ^ are ignored when matching). A false showUL turns off the underline without affecting the shortcut. Note that although the entire object label is searched for matching character to underline, the shortcut string itself is not searched, thus shortcut "Yy" for label "Yes" will result in the underlining of Y while "yY" will not.

To set the bitmap to use for the bitmap button, the following routines can be used,

```
void fl_set_bitmapbutton_data(FL_OBJECT *ob, int w, int h,
                              unsigned char *bits)

void fl_set_bitmapbutton_file(FL_OBJECT *ob, const char *filename)
```

Similarly, to set the pixmap to use for the pixmap button, the following routines can be used

```
void fl_set_pixmapbutton_data(FL_OBJECT *ob, unsigned char **bits)

void fl_set_pixmapbutton_file(FL_OBJECT *ob, const char *filename)

void fl_set_pixmapbutton_pixmap(FL_OBJECT *ob, Pixmap id, Pixmap mask)
```

In the first routine, you `#include` the pixmap file into you source code and use the pixmap definition data (an array of char pointers) directly. In the second routine, the filename that contains the pixmap defination is used to specify the pixmap. The last routine assumes that you've already have X Pixmap resource IDs for the pixmap you want to use. Note that these routines do not free the pixmaps already associated with the button. To free the pixmaps, use the following routine

```
void fl_free_pixmapbutton_pixmap(FL_OBJECT *ob);
```

This function frees the pixmap and mask together with all the colors they allocated.

To get the pixmap that is currently being displayed, use the following routine

```
Pixmap fl_get_pixmapbutton_pixmap(FL_OBJECT *ob,
                                  Pixmap &pixmap, Pixmap &mask)
```

Pixmaps are by default displayed centered inside the bounding box. However, this can be changed using the following routine

```
void fl_set_pixmapbutton_align(FL_OBJECT *ob, int align,
                         int xmargin, int ymargin)
```

where `align` is the same as that used for labels. See Section 3.11.3 for a list. `xmargin` and `ymargin` are extra margins to leave in addition to the object border width. Note that although you can place a pixmap outside of the bounding box, it probably is not a good idea.

Finally there is routine that can be used to disable the focus outline with a false `flag`

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

See also Section 15.6 for pixmap color and transparency handling.

Attributes

For normal buttons color1 controls the normal color and color2 the color when pushed. For lightbuttons color1 is the color of the light when off and color2 the color when on. For round buttons, color1 is the color of the circle and color2 the color of the circle that is placed inside it when pushed. For round3dbutton, color1 is the color of the inside of the circle and color2 the color of the embedded circle. For bitmapbuttons, color1 is the normal box color (or bitmap background if nobox) and color2 is used to indicate the focus color. The foreground color of the bitmap is controlled by label color. For scrollbutton, col1 is the overall boundbox color (if not NO_BOX), col2 is the arrow color. The label of the scrollbutton must be of a string of a number between 1 – 9 (except 5), indicating the arrow direciton like the numerical key pad. The label can have an optional prefix # to indicate uniform scaling. For example, a label "#9" indicates the arrow should be pointing up-right and the arrow has the identical width and height regardless the overall bounding box size.

Remarks

See all demo programs, in particular pushbutton.c and buttonall.c, for the use of buttons.

Chapter 17

Valuator objects

17.1 Slider

Short description

Sliders are useful for letting the user indicate a value between some fixed bounds. Both horizontal and vertical sliders exist. They have a minimum, maximum and current value (all floats). The user can change this value by shifting the slider with the mouse. Whenever the value changes, this is reported to the application program.

Adding an object

To add a slider to a form use

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

or

```
FL_OBJECT *fl_add_valslider(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 by default placed below the slider. The second type of slider displays its value above or to the left of the slider.

Types

The following types of sliders are available:

`FL_VERT_SLIDER`	A vertical slider.
`FL_HOR_SLIDER`	A horizontal slider.

`FL_VERT_FILL_SLIDER`	A vertical slider, filled from the bottom.
`FL_HOR_FILL_SLIDER`	A horizontal slider, filled from the left.
`FL_VERT_NICE_SLIDER`	A nice looking vertical slider.
`FL_HOR_NICE_SLIDER`	A nice looking horizontal slider.
`FL_VERT_BROWSER_SLIDER`	A different looking vertical slider.
`FL_HOR_BROWSER_SLIDER`	A different looking horizontal slider.

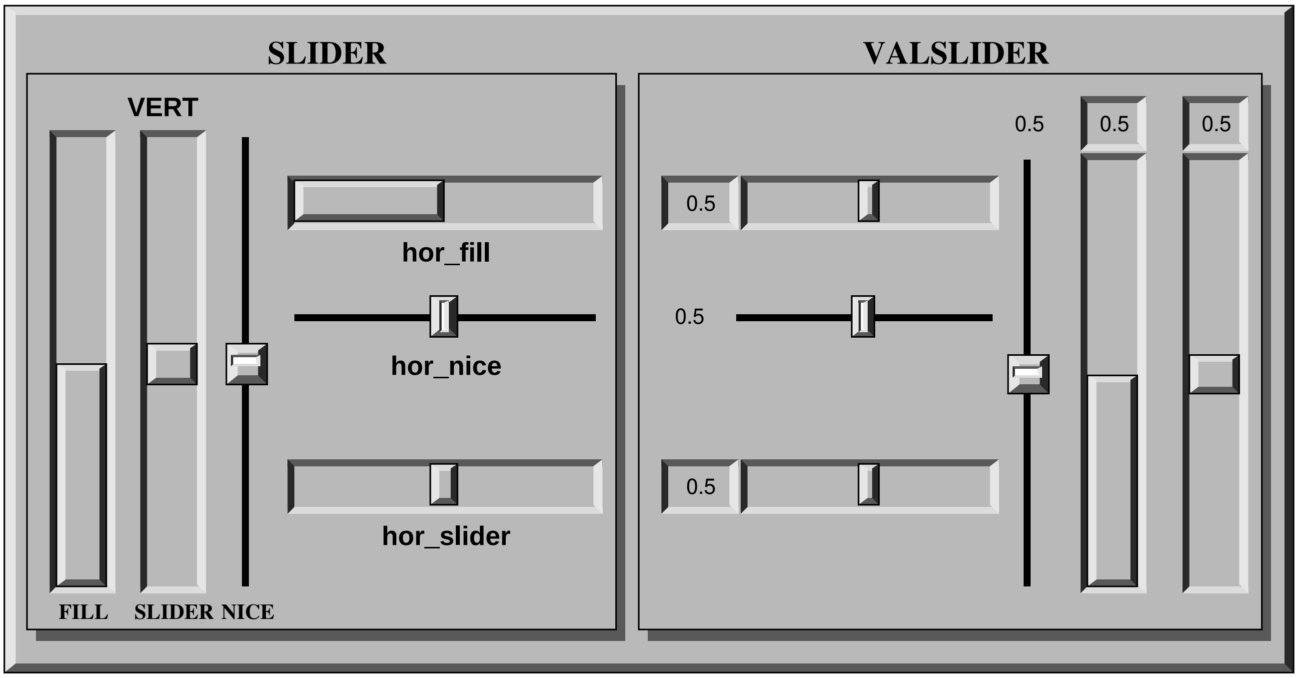

Figure 17.1: All sliders

Interaction

Whenever the user changes the value of the slider using the mouse, the slider is returned (or the callback called) by the interaction routines. The slider position is changed by moving the mouse inside the slider area. For fine control, hold down the left or right <SHIFT> key while moving the slider. Depending on the object size (pixels) and the slider value range, dragging the sliding bar might not always get the value you want, even with the <SHIFT> fine control, if the range is larger than the number of pixels (for example, if you use 100 pixels to represent 150 values, no matter how you control the motion, you will not get all the values). In these situations, you should use the right or middle mouse button with appropriate increments (see `fl_set_slider_increment()` below). The interaction with increment is that it updates the value first (as opposed to reading off the pixel position) then maps the value back into pixel position, thus all values of multiple increments are obtainable.

In some applications you might not want the slider to be returned all the time. To change the default, call the following routine:

```
void fl_set_slider_return(FL_OBJECT *obj, int when)
```

where parameter `when` can be one of the four values

`FL_RETURN_END_CHANGED` return at end (mouse release) if value is changed (since last return).

`FL_RETURN_CHANGED` return whenever the slider value is changed.

`FL_RETURN_END` return at end (mouse release) regardless if the value is changed or not.

`FL_RETURN_ALWAYS` return all the time. Not very useful.

See demo `objreturn.c`for an example use of this.

Other routines

To change the value and bounds of a slider use the following routines

```
void fl_set_slider_value(FL_OBJECT *obj, double val)

void fl_set_slider_bounds(FL_OBJECT *obj, double min, double max)
```

By default, the minimum value is 0.0, the maximum is 1.0 and the value is 0.5. For vertical sliders, `min` and `max` indicate, respectively, the values at the top and the bottom of the sliders, thus `max > min` needs not be observed.

To obtain the current value or bounds of a slider use

```
double fl_get_slider_value(FL_OBJECT *obj)

void fl_get_slider_bounds(FL_OBJECT *obj, double *min, double *max)
```

In a number of situations you would like slider values to be rounded to some values, e.g. to integer values. To this end use the routine

```
void fl_set_slider_step(FL_OBJECT *obj, double step)
```

After this call slider values will be rounded to multiples of `step`. Use the value 0.0 to stop rounding.

By default, if mouse is pressed below or above the the sliding bar, the sliding bar jumps to the location where the mouse is pressed. You can, however, use the following routine to change this default so the jumps are made is discrete increments:

```
void fl_set_slider_increment(FL_OBJECT *obj, double lj, double rj)
```

where `lj` indicates how much to jump if the left mouse button is pressed and `rj` indicates how much to increment if rigth/middle mouse buttons pressed. This routine can be used if finer control of the slider value is needed or assigning different meaning to different mouse buttons. For example, for the slider in the browser class, the left mouse jump is made to be one page and right jump is made to be one line.

To obtain the current increment, use the following routine

```
void fl_get_slider_increment(FL_OBJECT *obj, float *lj, float *rj)
```

Attributes

Never use `FL_NO_BOX` as boxtype for a slider. For `FL_VERT_NICE_SLIDER`s and `FL_HOR_NICE_SLIDER`s one best uses a `FL_FLAT_BOX` in the color of the background to get the nicest effect. Color1 controls the color of the background of the slider, color2 the color of the slider itself.

You can control the size of the slider inside the box using the routine

```
void fl_set_slider_size(FL_OBJECT *obj, double size)
```

`size` should be a float between 0.0 and 1.0. The default is `FL_SLIDER_WIDTH` = 0.10 for regular sliders and 0.15 for browser sliders. With size=1.0, the slider covers the box completely and can no longer be moved. This function does not apply to `NICE_SLIDER` and `FILL_SLIDER`.

The routine

```
void fl_set_slider_precision(FL_OBJECT *obj, int prec)
```

sets the precision with which the value of the slider is shown. This only applies to sliders showing their value.

By default, the value shown by valslider is the slider value in floating point format. You can override the default by registering a filter function using the following routine

```
void fl_set_slider_filter(FL_OBJECT *obj,
                          const char *(*filter)(FL_OBJECT *,
                                                double value,
                                                int prec));
```

where `value` and `prec` are the slider value and precision respectively. The filter function `filter` should return a string that is to be shown. The default filter is equivalent to the following

```
const char *filter(FL_OBJECT *ob, double value, int prec)
{
    static char buf[32];
    sprintf(buf, "%.*f", prec, value);
    return buf;
}
```

Remarks

See the demo program `demo05.c` for an example of the use of sliders. See demos `sldsize.c` and `sliderall.c` for the effect of setting slider sizes and the different types of sliders.

Although all function prototypes would seem to indicate that sliders have a resolution of a double, it is not true. All internal calculations are done with float precision.

17.2 Scrollbars

Short description

Scrollbars are similar to sliders (as a matter of fact, scrollbars are made with sliders and scrollbuttons), and useful in letting the user indicate a value between some fixed bounds. Both horizontal and vertical scrollbars exist. They have a minimum, maximum and current value (all floats). The user can change this value by dragging the sliding bar with the mouse or press the scroll buttons. Whenever the value changes, it is reported to the application program via the callback function.

Adding an object

To add a scrollbar to a form use

```
FL_OBJECT *fl_add_scrollbar(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 by default placed below the scrollbar.

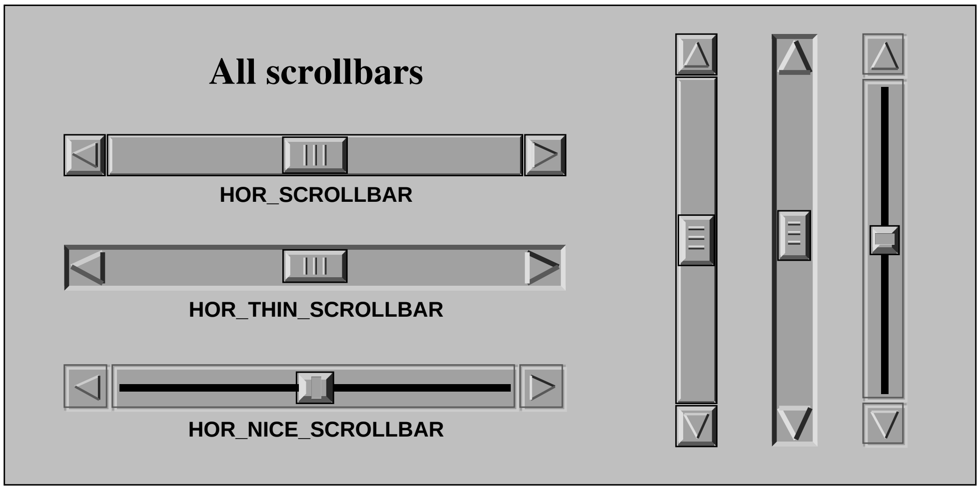

Figure 17.2: All Scrollbars

Types

The following types of scrollbar are available:

FL_VERT_SCROLLBAR A vertical scrollbar.

FL_HOR_SCROLLBAR A horizontal scrollbar.

FL_VERT_THIN_SCROLLBAR A different looking vertical scrollbar.

FL_HOR_THIN_SCROLLBAR A different looking horizontal scrollbar.

FL_VERT_NICE_SCROLLBAR A vertical scrollbar using NICE_SLIDER.

FL_HOR_NICE_SCROLLBAR A horizontal scrollbar using NICE_SLIDER.

FL_VERT_PLAIN_SCROLLBAR Similar to THIN_SCROLLBAR.

FL_HOR_PLAIN_SCROLLBAR Similar to THIN_SCROLLBAR.

Interaction

Whenever the user changes the value of the scrollbar using the mouse, the scrollbar's callback is called by the main loop. The scrollbar position is changed by moving the mouse inside the scrollbar area. For fine control, hold down the left or right <SHIFT> key while moving the slider.

In some applications you might not want the scrollbar to be returned all the time. To change the default, call the following routine:

```
void fl_set_scrollbar_return(FL_OBJECT *obj, int when)
```

where parameter `when` can be one of the following four values

FL_RETURN_END_CHANGED return at end (mouse release) if value is changed (since last return).

FL_RETURN_CHANGED return whenever the scrollbar value is changed.

FL_RETURN_END return at end (mouse release) regardless if the value is changed or not.

FL_RETURN_ALWAYS return all the time. Not very useful.

See demo `objreturn.c`for an example use of this.

Other routines

To change the value and bounds of a scrollbar use the following routines

```
void fl_set_scrollbar_value(FL_OBJECT *obj, double val)

void fl_set_scrollbar_bounds(FL_OBJECT *obj, double min, double max)
```

By default, the minimum value is 0.0, the maximum is 1.0 and the value is 0.5. For vertical scrollbars, `min` and `max` indicate, respectively, the value at the top and the bottom of the scrollbars, thus `max > min` needs not be observed.

To obtain the current value and bounds of a scrollbar use

```
double fl_get_scrollbar_value(FL_OBJECT *obj)

void fl_get_scrollbar_bounds(FL_OBJECT *obj, double *min, double *max)
```

In a number of situations you would like scrollbar values to be rounded to some values, e.g. to integer values. To this end use the routine

```
void fl_set_scrollbar_step(FL_OBJECT *obj, double step)
```

After this call scrollbar values will be rounded to multiples of `step`. Use the value 0.0 to stop rounding. This should not be confused with the increment/decrement value when the scroll buttons are pressed. Use `fl_set_scrollbar_increment()` to change the increment value.

By default, if mouse is pressed below or above the the sliding bar, the sliding bar jumps to the location where the mouse is pressed. You can, however, use the following routine to change this default so the jumps are made is discrete increments:

```
void fl_set_scrollbar_increment(FL_OBJECT *obj, double lj, double rj)
```

where `lj` indicates how much to increment if the left mouse button is pressed and `rj` indicates how much to jump if right/middle mouse button pressed. For example, for the scrollbar in the browser class, the left mouse jump is made to be one page and right/middle mouse jump is made to be one line. The increment (decrement) value when the scrollbuttons are pressed is set to the value of the right jump.

To obtain the current increment settings, use the following routine

```
void fl_get_scrollbar_increment(FL_OBJECT *ob, float *lj, float *sj)
```

Attributes

Never use `FL_NO_BOX` as boxtype for a scrollbar. For `FL_VERT_NICE_SCROLLBAR`s and `FL_HOR_NICE_SCROLLBAR`s one best uses a `FL_FLAT_BOX` in the color of the background to get the nicest effect. Color1 controls the color of the background of the scrollbar, color2 the color of the sliding bar itself.

You can control the size of the sliding bar inside the box using the routine

```
void fl_set_scrollbar_size(FL_OBJECT *obj, double size)
```

`size` should be a float between 0.0 and 1.0. The default is `FL_SLIDER_WIDTH` = 0.15 for all scrollbars With size=1.0, the scrollbar covers the box completely and can no longer be moved. This function does not apply to `NICE_SCROLLBAR`.

Remarks

See the demo program `scrollbar.c` for an example of the use of scrollbars.

Although all function prototypes would indicate that scrollbars have a resolution of a double, it is not true. All internal calculations are done with float precision.

Also note that the `get` routines take pointers to floats as parameters while the `set` routines take doubles as parameters. It is perfectly legal to pass floats as doubles. The reason for this inconsistency is to work around some (buggy) C++ compilers that always widen function parameters, causing mismatches between the compiled library and the application program.

17.3 Dial

Short description

Dial objects are dials that the user can put in a particular position using the mouse. They have a minimum, maximum and current value (all floats). The user can change this value by turning the dial with the mouse. Whenever the value changes, this is reported to the application program.

Adding an object

To add a dial to a form use

```
FL_OBJECT *fl_add_dial(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 by default placed below the dial.

Types

The following types of dials are available:

`FL_NORMAL_DIAL`	A dial with a knob indicating the position.
`FL_LINE_DIAL`	A dial with a line indicating the position.
`FL_FILL_DIAL`	The area between initial and current is filled.

Interaction

By default, the dial value is returned to the application when the user releases the mouse. It is possible to change this behavior using the following routine

```
void fl_set_dial_return(FL_OBJECT *obj, int how_return)
```

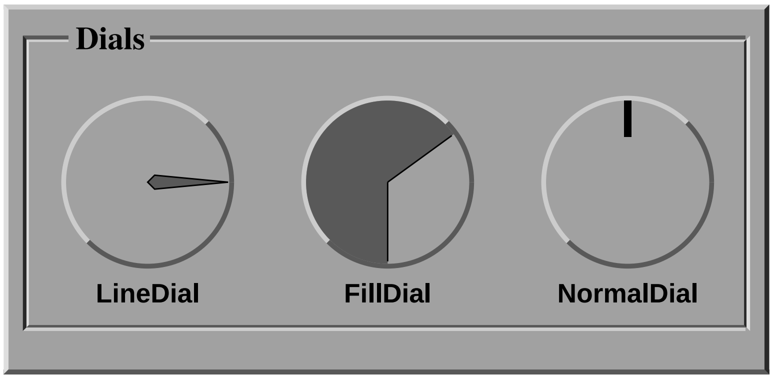

Figure 17.3: Types of dials

where `how_return` can be one of the following

FL_RETURN_END_CHANGED Return at end (mouse release) and only if the dial value is changed. The Default.

FL_RETURN_CHANGED Return whenever the dial value is changed.

FL_RETURN_END Return at the end regardless if the dial value is changed or not.

Other routines

To change the value of the dial use

```
void fl_set_dial_value(FL_OBJECT *obj, double val)

void fl_set_dial_bounds(FL_OBJECT *obj, double min, double max)
```

By default, the minimum value is 0.0, the maximum is 1.0 and the value is 0.5. To obtain the current values of the dial use

```
double fl_get_dial_value(FL_OBJECT *obj)

void fl_get_dial_bounds(FL_OBJECT *obj, double *min, double *max)
```

Sometimes, it might be desirable to limit the angular range a dial can take or choose an angle other than 0 to represent the minimum value. For this purpose, use the following routine

```
void fl_set_dial_angles(FL_OBJECT *ob, double thetai, double thetaf)
```

where `thetai` maps to the minimum value of the dial and `thetaf` maps to the maximum value of the dial. The angles are relative to the *origin* of the dial, which is by default at 6 o'clock (270^{o}

from 3 o'clock) and rotates clock-wise. By default, the minimum angle is 0 and the maximum angle is 360.

By default, crossing from 359.9 to 0 or from 0 to 359.9 is not allowed. To allowing crossing, use the following routine

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

In a number of situations you might want dial values to be rounded to some values, e.g. to integer values. To this end use the routine

```
void fl_set_dial_step(FL_OBJECT *obj, double step)
```

After this call dial values will be rounded to multiples of `step`. Use the value 0.0 to stop rounding.

By default, clock-wise rotation increases the dial value. To change, use the following routine

```
void fl_set_dial_direction(FL_OBJECT *obj, int dir)
```

where `dir` can be either `FL_DIAL_CCW` or `FL_DIAL_CW`.

Attributes

You can use any boxtype you like, but the final dial face always appears to be circular although certain correlation between the requested boxtype and actual boxtype exists (for example, `FL_FRAME_BOX` is translated into a circular frame box.)

Color1 controls the color of the background of the dial, color2 the color of the knob or the line or the fill color.

Remarks

The resolution of a dial is about 0.2 degrees, i.e., there are only about 2000 steps per 360 degrees and depending on the size of the dial, it is typically less.

The dial is always drawn with a circular box. If you specify a `FL_UP_BOX`, a `FL_OVAL3D_UPBOX` will be used.

See the demo program `ldial.c`, `ndial.c` and `fdial.c` for examples of the use of dials.

17.4 Positioner

Short description

A positioner is an object in which the user can indicate a position with an x- and a y-coordinate. It displays a box with a cross-hair cursor in it. Clicking the mouse inside the box changes the position of the cross-hair cursor and, hence, the x- and y-values.

Adding an object

A positioner can be added to a form using the call

```
FL_OBJECT *fl_add_positioner(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 placed below the box by default.

Types

The following types of positioner exist:

FL_NORMAL_POSITIONER	Cross-hair inside a box.
FL_OVERLAY_POSITIONER	Cross-hair inside a box, but moves in XOR mode.

Interaction

The user changes the setting of the positioner using the mouse inside the box. Whenever the values change, the object is returned by the interaction routines.

In some applications you only want the positioner to be returned to the application program when the user releases the mouse, i.e., not all the time. To achieve this call the routine

```
void fl_set_positioner_return(FL_OBJECT *obj, int always)
```

Set `always` to 0 to achieve this goal.

Other routines

To set the value of the positioner and the boundary values use the routines:

```
void fl_set_positioner_xvalue(FL_OBJECT *obj, double val)

void fl_set_positioner_xbounds(FL_OBJECT *obj, double min, double max)

void fl_set_positioner_yvalue(FL_OBJECT *obj, double val)

void fl_set_positioner_ybounds(FL_OBJECT *obj, double min, double max)
```

By default the minimum values are 0.0, the maximum values are 1.0 and the values are 0.5. For ybounds, `min` and `max` should be taken to mean the value at the bottom and value at the top of the positioner.

To obtain the current values of the positioner use

```
double fl_get_positioner_xvalue(FL_OBJECT *obj)

void fl_get_positioner_xbounds(FL_OBJECT *obj, double *min, double *max)

double fl_get_positioner_yvalue(FL_OBJECT *obj)

void fl_get_positioner_ybounds(FL_OBJECT *obj, double *min, double *max)
```

In a number of situations you would like positioner values to be rounded to some values, e.g. to integer values. To this end use the routines

```
void fl_set_positioner_xstep(FL_OBJECT *obj, double step)

void fl_set_positioner_ystep(FL_OBJECT *obj, double step)
```

After these calls positioner values will be rounded to multiples of `step`. Use the value 0.0 to stop rounding.

Sometimes, it makes more sense for a positioner to have an icon/pixmap as the background that represents a minified version of the area where positioner's values apply. Type `OVERLAY_POSITIONER` is specifically designed for this by drawing the moving cross-hair in XOR mode as not to erase the background. A typical creation procedure might look something like the following

```
obj = fl_add_pixmap(FL_NORMAL_PIXMAP, x, y, w, h, label);
   fl_set_pixmap_file(obj, iconfile);
pos = fl_add_positioner(FL_OVERLAY_POSITIONER, x, y, w, h, label);
```

Of course, you can overlay this type of positioner on objects other than a pixmap. See demo program `positionerXOR.c` for an example.

Attributes

Never use `FL_NO_BOX` for a positioner. Color1 controls the color of the box, color2 the color of the cross-hair.

Remarks

A demo can be found in `positioner.c`.

17.5 Counter

Short description

A counter provides a different mechanism for the user to indicate a value. In consists of a box displaying the value and four buttons two at the left and two at the right side. The user can press these buttons to change the value. The extreme buttons make the value change fast, the other buttons make it change slowly. As long as the user keeps his mouse pressed, the value changes.

Adding an object

To add a counter to a form use

```
FL_OBJECT *fl_add_counter(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 by default placed below the counter.

Types

The following types of counters are available:

`FL_NORMAL_COUNTER`	A counter with two buttons on each side.
`FL_SIMPLE_COUNTER`	A counter with one button on each side.

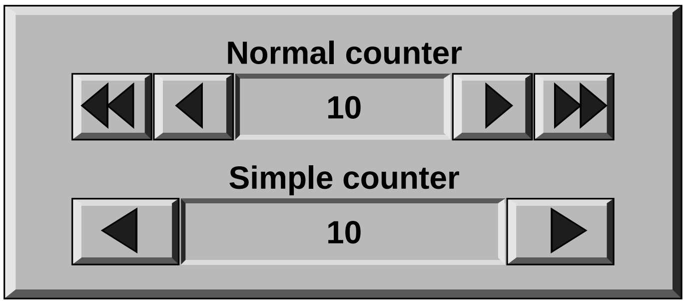

Figure 17.4: Counter types

Interaction

The user changes the value of the counter by keeping his mouse pressed on one of the buttons. Whenever he releases the mouse the counter is returned to the application program.

In some applications you might want the counter to be returned to the application program whenever the value changes. To this end, the following routine is available

```
void fl_set_counter_return(FL_OBJECT *obj, int how)
```

where `how` can be either `FL_RETURN_END_CHANGED` (the default) or `FL_RETURN_CHANGED`.

Other routines

To change the value of the counter use the routines

```
void fl_set_counter_value(FL_OBJECT *obj, double val)

void fl_set_counter_bounds(FL_OBJECT *obj, double min, double max)

void fl_set_counter_step(FL_OBJECT *obj, double small, double large)
```

The first routine sets the value (default is 0), the second routine sets the minimum and maximum values that the counter will take (default -1000000 and 1000000) and the third routine sets the sizes of the small and large steps (default 0.1 and 1). (For simple counters only the small step is used.)

For conflicting settings, bounds take precedence over value, i.e., if setting a value that is outside of the current bounds, it is clamped.

To obtain the current value of the counter use

```
double fl_get_counter_value(FL_OBJECT *obj)
```

To obtain the current bounds and steps, use the following functions

```
void fl_get_counter_bounds(FL_OBJECT *obj, double *min, double *max)

void fl_get_counter_step(FL_OBJECT *obj, float *small, float *large)
```

To set the precision (number of digits after the dot) with which the counter value is displayed use the routine

```
void fl_set_counter_precision(FL_OBJECT *obj, int prec)
```

By default, the value shown is the counter value in floating point format. You can override the default by registering a filter function using the following routine

```
void fl_set_counter_filter(FL_OBJECT *obj,
                           const char *(*filter)(FL_OBJECT *,
                                                 double value,
                                                 int prec));
```

where `value` and `prec` are the counter value and precision respectively. The filter function `filter` should return a string that is to be shown. Note that the default filter is equivalent to the following

```
const char *filter(FL_OBJECT *ob, double value, int prec)
{
    static char buf[32];
    sprintf(buf, "%.*f",prec,value);
    return buf;
}
```

Attributes

Never use FL_NO_BOX as boxtype for a counter. Color1 controls the color of the background of the counter, color2 the color of the arrows in the counter.

Remarks

Although function prototypes give the impression that a counter has a resolution of a double, this is not true. Internal calculation is done entirely in float precision. The reason for using doubles as function parameters is to workaround bugs in some C++ compilers that always promote floats to doubles for C functions.

See `counter.c` for an example of the use of counters.

Chapter 18

Input objects

18.1 Input

Short description

It is often required to obtain textual input from the user, e.g. a file name, some fields in a database, etc. To this end input fields exist in the **Forms Library**. An input field is a field that can be edited by the user using the keyboard.

Adding an object

To add an input field to a form you use the routine

```
FL_OBJECT *fl_add_input(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 by default placed in front of the input field.

Types

The following types of input fields exist:

`FL_NORMAL_INPUT`	Any type of text can be typed into this field.
`FL_FLOAT_INPUT`	Only a float value can be typed in (e.g. -23.2e12).
`FL_INT_INPUT`	Only an integer value can be typed in (e.g. -86).
`FL_DATE_INPUT`	Only a date (MM/DD/YY) can be typed in.
`FL_MULTILINE_INPUT`	An input field allowing for multiple lines.
`FL_SECRET_INPUT`	A normal input field that does not show the text.
`FL_HIDDEN_INPUT`	A normal input field but invisible.

A normal input field can contain one line of text, to be typed in by the user. A float input field can only contain a float number. If the user tries to type in something other than a float, it is not shown and the bell is sounded. Similarly, an int input field can only contain an integer number and a date input field can only contain a valid date (see below). A multi-line input field can contain multiple lines of text. A secret input field works like a normal input field but the text is not shown. Only the cursor is shown which does move while text is being entered. This can be used for getting passwords, for example. Finally, a hidden input field is not shown at all but does collect text for the application program to use.

Interaction

Whenever the user presses the mouse inside an input field a cursor will appear in it (and the field will change color to indicate input focus). Further input will be directed to this field. The user can use the following keys (as in *emacs*(1)) to edit or move around inside the input field:

delete previous char	<DELETE>
delete next char	<CNTRL> `D`
delete previous word	<META> <DELETE>, <CNTRL> `W`
delete next word	<META> `d`
delete to end of line	<CNTRL> `k`
backspace	<CNTRL> `H`
to beginning of line	<CNTRL> `A`,<SHIFT><LEFT>
to end of line	<CNTRL> `E`, <SHIFT><RIGHT>
char backward	<CNTRL> `B`, <LEFT>
char forward	<CNTRL> `F`, <RIGHT>
next line	<CNTRL> `N`, <DOWN>
previous line	<CNTRL> `P`, <UP>
next page	<PAGEDOWN>
previous page	<PAGEUP>
word backward	<META> `b`
word forward	<META> `f`
beginning of field	<META> <, <HOME>, <SHIFT><UP>
end of field	<META> >, <END>, <SHIFT><DOWN>
clear input field	<CNTRL> `U`
paste	<CNTRL> `y`

It is possible to remap the the bindings, see later for details.

There are three ways to select part of the input field. Dragging, double-click and triple-click. Double-click selects the word the mouse is on and triple-click selects the entire line the mouse is on. The selected part of the input field is removed when the user types the <BACKSPACE> key or replaced by what the user types in. Also the cursor can be placed at different positions in the input field using the mouse.

One additional property of selecting part of the text field is that if the selection is done with the leftmouse the selected part becomes the primary (`XA_PRIMARY`) selection of the X Selection mechanism, thus other applications, e.g., `xterm`, can request this selection. Conversely, the cutbuffers from other applications can be pasted into the input field. Use the middle mouse for pasting. Note <CNTRL> `y` only pastes the cutbuffer generated by <CNTRL> `k` and is not related to the X Selection mechanism, thus it only works within the same application.

When the user presses the <TAB> key the input field is returned to the application program and the input focus is directed to the next input field. This also happens when the user presses the <RETURN> key but only if the form does not contain a return button. The order which input field gets the focus when the <TAB> is pressed is the same as the order the input field is added to the form. From within **Form Designer**, using the raising function arrange (re-arrange) the focus order. See Part II Section 10.6 for details.

This (<TAB> and <RETURN>) does not work for multi-line input fields where <RETURN> key is used to separate lines and <TAB> is a legitimate input character (not currently handled though). Also when the user picks a new input field with the mouse, the current input object is returned.

The above mechanism is the default behavior of an input field. Depending on the application, other options might be useful. To change the precise condition for the object to be returned (or equivalently the callback invoked), the following function can be used:

```
void fl_set_input_return(FL_OBJECT *obj, int when)
```

Where `when` can take one of the following values:

`FL_RETURN_END_CHANGED` The default. Callback is called at the end if the field is modified.

`FL_RETURN_CHANGED` Invoke the callback function whenever the field is changed.

`FL_RETURN_END` Invoke the callback function at the end regardless if the field is modified or not.

`FL_RETURN_ALWAYS` Invoke the callback function upon each keystroke.

See demo `objreturn.c` for an example use of this.

There is a routine that can be used to limit the number of characters per line for `NORMAL_INPUT`

```
void fl_set_input_maxchars(FL_OBJECT *ob, int maxchars);
```

To reset the limit to infinite, set `maxchars` to 0.

Although an input with `FL_RETURN_ALWAYS` attributes can be used in combination with the callback function to check the validity of characters that are entered into the input field, use of the following is typically more appropriate

```
typedef int (*FL_INPUTVALIDATOR)(FL_OBJECT *ob,
            const char *old, const char *cur, int c);
FL_INPUTVALIDATOR fl_set_input_filter(FL_OBJECT *ob,
                  FL_INPUTVALIDATOR filter);
```

The filter function is called whenever a new (regular) character is entered. `old` is the string in the input field before the newly typed character `c` is merged into `cur`. If the new character is not an acceptable character for the input field, the `filter` function should return `FL_INVALID` otherwise

FL_VALID. If FL_INVALID is returned, the new character is discarded and the input field remains unmodified. The function returns the old filter. Unlike the built-in filters, keyboard bell is not sound when FL_INVALID is received. To sound the bell, return FL_INVALID|FL_RINGBELL.

This still leaves the possibility that the input is valid for every character entered, but the string is invalid for the field because it is incomplete. For example, 12.0e is valid for a float input field for every character typed, but the final string is not a valid floating point. To guard again this, the filter function is called just prior to returning the object with c set to zero. If the validator returns FL_INVALID, the object is not returned to the application program, but input focus can change to the next input field. If the return value is FL_INVALID|FL_RINGBELL, keyboard bell is sound and the object is not returned to the application program. Further, the input focus is not changed.

To facilitate specialized input fields using validators, the following validator dependent routines are available

```
void fl_set_input_format(FL_OBJECT *ob, int attrib1, int attrib2)

void fl_get_input_format(FL_OBJECT *ob, int *attrib1, int *attrib2)
```

These two routines more or less provide a means for the validator to store and retrieve some information about user preference or other state dependent information. attrib1 and attrib2 can be any validator defined variables. For the built-in class, only DATE_INPUT utilizes these to store the date format: for attrib1, it can take FL_INPUT_MMDD or FL_INPUT_DDMM and attrib2 is the separator between month and day. For example, to set the date format to dd/mm, use the following

```
void fl_set_input_format(ob, FL_INPUT_DDMM,'/')
```

For the built-in DATE_INPUT the default is FL_INPUT_MMDD and the separator is '/'. There is no limit on the year other than it must be an integer and appear after month and day.

Other routines

Note that the label is not the default text in the input field.

To set the contents of the input field use the routine

```
void fl_set_input(FL_OBJECT *obj, const char *str)
```

There is very limited check for the validity of str for the input field. Use an empty string the clear an input field.

Setting the content of an input field does not trigger object event, i.e., the object callback is not called. In some situations you might want to have the callback invoked. For this, use the following the routine

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

To obtain the string in the field (when the user has changed it) use:

```
const char *fl_get_input(FL_OBJECT *obj)
```

This function returns a char pointer for all input types. Thus for numerical input types, *atoi*(3), *atof*(3) or *sscanf*(3) should be used to convert the string to the proper data type you need. For multiline input, the returned pointer points to the entire content with possibly embedded newlines. The application should not modify the content pointed to by the returned pointer, which points to the internal buffer.

To select or deselect the current input or part of it, the following two routines can be used

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

void fl_set_input_selected_range(FL_OBJECT *ob, int start, int end)
```

where `start` and `end` are measured in characters. When `start` is 0 and `end` equals the number of characters in the string, `fl_set_input_selected()` makes the entire input field selected. However, there is a subtle difference between this routine and `fl_set_input_selected()` with `flag==1`. `fl_set_input_selected()` always places the cursor at the end of the string while `fl_set_input_selected_range()` places the cursor at the beginning of the selection.

To obtain the currently selected range, either selected by the application or by the user, use the following routine

```
const char *fl_get_input_selected_range(FL_OBJECT *ob,
                                        int *start, int *end)
```

Where `start` and `end`, if not null, are set to the begining and end position of the selected range, measured in characters. For example, if `start` is 5, and `end` is 7, it means the selection starts at character 6 (`str[5]`) and ends before character 8 (`str[7]`), so a total of two characters are selected (i.e., character 6 and 7). The function returns the selected string. If there is currently no selection, the function returns null and both `start` and `end` are set to -1. Note that the char pointer returned by the function points to (kind of) a static buffer, and will be overwritten by the next call.

It is possible to obtain the cursor position using the following routine

```
int fl_get_input_cursorpos(FL_OBJECT *ob, int *xpos, int *ypos)
```

The function returns the cursor position measured in number of characters (including newline characters) in front of the cursor. If the input field does not have input focus (thus does not have a cursor), the function returns -1. Upon function return, `ypos` is set to the line number (starting from 1) the cursor is on and `xpos` set to the number of characters in front of the cursor measured from the beginning of the current line, i.e., `ypos`. If the input field does not have input focus, the `xpos` is set to -1.

It is possible to move the cursor within the input field programmatically using the following routine

```
void fl_set_input_cursorpos(FL_OBJECT *ob, int xpos, int ypos)
```

where `xpos` and `ypos` are measured in characters (lines). E.g., given the input field, `"an arbitrary string"`, the call `fl_set_input_cursorpos(ob, 4, 1)` places the the cursor after the first character <A> in `arbitrary`.

Shortcut keys can be associated with an input field to switch input focus. To this end, use the following routine

```
void fl_set_input_shortcut(FL_OBJECT *obj, const char *sc, int showit)
```

By default, if a `MULTILINE_INPUT` field contains more text than that can be shown, scrollbars will appear with which the user can scroll the text around horizontally or vertically. To change this default, use the following routines

```
void fl_set_input_hscrollbar(FL_OBJECT *ob, int how)

void fl_set_input_vscrollbar(FL_OBJECT *ob, int how)
```

where `how` can be one of the following values

`FL_AUTO` The default. Shows the scrollbar only if needed.

`FL_ON` Always shows the scrollbar.

`FL_OFF` No scrollbar is shown.

Note however, turning off scrollbars for an input field does not turn off scrolling, the user can still scroll the field using cursor and other keys.

To completely turn off scrolling for an input field (for both multiline and single line input field), use the following routine with a false `flag`

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

There are also routines that can scroll the input field programmatically. To scroll vertically (for `MULTILINE_INPUT` only), use the following routine

```
void fl_set_input_topline(FL_OBJECT *obj, int line)
```

where `line` is the new top line (starting from 1) in the input field.

To scroll horizontally, use the following routine

```
void fl_set_input_xoffset(FL_OBJECT *ob, int pixels)
```

where `pixels`, a positive number, indicates how many pixels to scroll to the left relative to the nominal position of the text lines.

To obtain the current xoffset, use the following function

```
int fl_get_input_xoffset(FL_OBJECT *ob)
```

To obtain the number of lines in the input field, use the following routine

```
int fl_get_input_numberoflines(FL_OBJECT *ob)
```

To obtain the current topline in the input field, use

```
int fl_get_input_topline(FL_OBJECT *ob)
```

To obtain the number of lines that fit inside the input box, use the following routine

```
int fl_get_input_screenlines(FL_OBJECT *ob)
```

Attributes

Never use `FL_NO_BOX` as boxtype.

Color1 controls the color of the input field when it is not selected and color2 is the color when selected.

To change the color of the input text or the cursor use

```
void fl_set_input_color(FL_OBJECT *obj, int tcol, int ccol)
```

Here `tcol` indicates the color of the text and `ccol` is the color of the cursor.

By default, the scrollbar size is dependent on the initial size of the input box. To change the size of the scrollbars, use the following routine

```
void fl_set_input_scrollbarsize(FL_OBJECT *ob, int hh, int vw)
```

where `hh` is the horizontal scrollbar height and `vw` is the vertical scrollbar width in pixels.

The default scrollbar type is `THIN_SCROLLBAR`. There are two ways you can change the default. One way is to use `fl_set_defaults()` or `fl_set_scrollbar_type()` to set the application wide default (preferred); another way is to use `fl_get_object_component()` to get the object handle to the scrollbars and change the the object type forcibly. Although the second method of changing the scrollbar type is not recommended, the object handle obtained can be useful in changing the scrollbar colors etc.

As mentioned earlier, it is possible for the application program to change the default edit keymaps. The editing key assignment is held in a `FL_EditKeymap` structure defined as follows:

```
typedef struct
{
   long del_prev_char;         /* delete previous char    */
   long del_next_char;         /* delete next char        */
   long del_prev_word;         /* delete previous word    */
   long del_next_word;         /* delete next word        */

   long moveto_prev_line;     /* one line  up             */
   long moveto_next_line;     /* one line down            */
   long moveto_prev_char;     /* one char left            */
   long moveto_next_char;     /* one char right           */
   long moveto_prev_word;     /* one word left            */
   long moveto_next_word;     /* one word right           */
   long moveto_prev_page;     /* one page up              */
   long moveto_next_page;     /* one page down            */
   long moveto_bol;           /* move to begining of line */
   long moveto_eol;           /* move to end of line      */
   long moveto_bof;           /* move to begin of file    */
   long moveto_eof;           /* move to end of file      */

   long transpose;            /* switch two char positions*/
   long paste;                /* paste the edit buffer    */
   long backspace;            /* alias for del_prev_char  */
   long del_to_eol;           /* cut to end of line       */
   long del_to_bol;           /* cut to end of line       */
   long clear_field;          /* delete all               */
   long del_to_eos;           /* not implemented yet      */
} FL_EditKeymap;
```

To change the default edit keymaps, the following routine is available:

```
void fl_set_input_editkeymap(FL_EditKeymap *km)
```

with a filled or partially filled `FL_EditKeymap` structure. The unfilled members must be set to zero so the default mapping is retained. Change of edit keymap is global and affects all input field within the application.

Setting `km` to null restores the default.

All cursor keys (<LEFT>, <HOME> etc.) are reserved and their meanings hard-coded, thus can't be used in the mapping. For example, if you try to set `del_prev_char` to <HOME>, pressing the <HOME> key will *not* delete the previous character.

In filling the keymap structure, regular control characters (value < 32) and ASCII characters (< 128) should be given their ASCII codes (<CNTRL> `C` is 3 etc) and special characters their Keysyms (`XK_F1` etc). Control and special character combination is obtained by adding `FL_CONTROL_MASK` to the keysym. To specify meta add `FL_ALT_MASK` to the key value.

```
FL_EditKeymap ekm;

memset(ekm, 0 , sizeof(ekm));           /* zero struct    */
ekm.del_prev_char = 8;                  /* control-H      */
ekm.del_next_char = 127;                /* delete         */
```

```
ekm.del_prev_word = 'h'|FL_ALT_MASK;    /* meta-H         */
ekm.del_next_word = 127|FL_ALT_MASK;    /* meta-delete    */
ekm.moveto_bof = XK_F1;                 /* F1 to bof      */
ekm.moveto_eof = XK_F1|FL_CONTROL_MASK; /* cntl-F1 to eof */

fl_set_input_editkeymap(&ekm);
```

Remarks

Always make sure that the input field is high enough to contain a single line of text. If the field is not high enough, the text may get clipped, i.e., invisible.

See the program `demo06.c` for an example of the use of input fields. See `minput.c` for multi-line input fields. See `secretinput.c` for secret input fields and `inputall.c` for all input fields.

Chapter 19

Choice objects

19.1 Menu

Short description

Also menus can be added to forms. These menus can be used to let the user choose from many different possibilities. Each menu object has a box with a label in it in the form. Whenever the user presses the mouse inside the box (or moves the mouse on top of the box) a pop-up menu appears. The user can then make a selection from the menu.

Adding an object

To add a menu to a form use the routine

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

It shows a box on the screen with the label centered in it.

Types

The following types are available:

`FL_PUSH_MENU`	The menu appears when the user presses a mouse button on it.
`FL_PULLDOWN_MENU`	The menu appears when the user presses a mouse button on it.
`FL_TOUCH_MENU`	The menu appears when the user move the mouse inside it.

`PUSH_MENU` and `PULLDOWN_MENU` behaves in exactly the same way. The only difference is in the way they are drawn when the menu is active: `PUSH_MENU`'s menu appears to be an up_box casting a shadow while `PULLDOWN_MENU`'s is just an extension of the menu box (see Fig. 19.1).

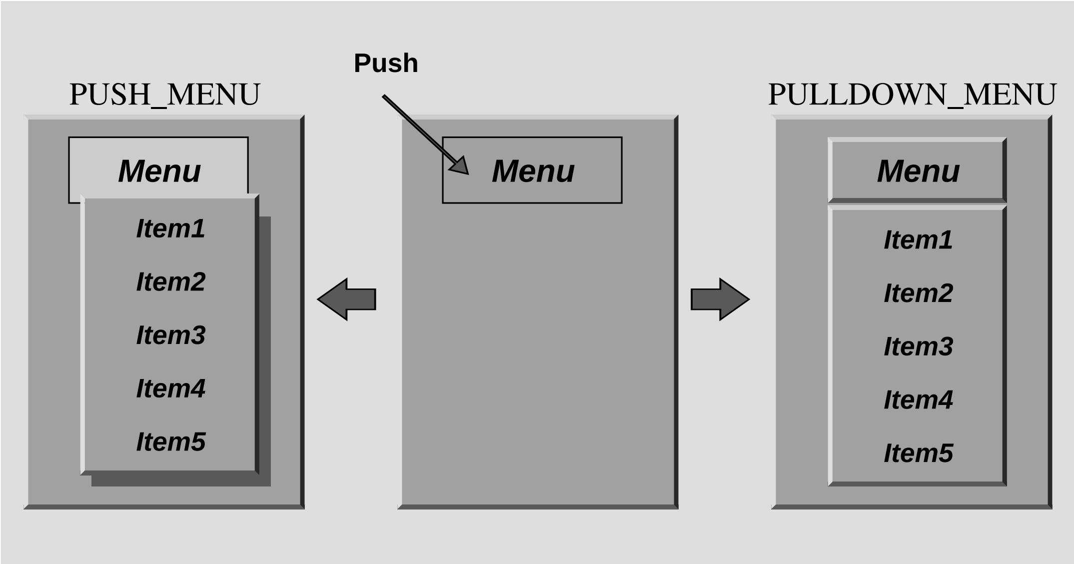

Figure 19.1: Menu types

Interaction

When the menu appears the user can make a selection using the left mouse button or make no selection by clicking outside the menu. When he makes a selection the menu object is returned by the interaction routines.

Other routines

To set the actual menu for a menu object, use the routine

```
void fl_set_menu(FL_OBJECT *obj, const char *menustr)
```

`menustr` describes the menu in the form used by XPopups (See Section 21.3. In short, it should contain the menu items, separated by a bar, e.g., "First|Second|Third". See Section 21.3 for special tags that can be used to indicate special attributes (radio, toggle and gray for example. Specifying values via %x is not supported). Whenever the user selects some menu item, the menu object is returned to the application program.

To find the actual menu item selected by the user use

```
int fl_get_menu(FL_OBJECT *obj)
```

When the first item is selected 1 is returned, for the second item 2, etc. If no item was selected -1 is returned.

You can also obtain the text of the item selected

```
const char *fl_get_menu_text(FL_OBJECT *obj)
```

To obtain the text of any item, use the following routine

```
const char *fl_get_menu_item_text(FL_OBJECT *obj, int n)
```

To obtain the total number of menu items, use the following function

```
int fl_get_menu_maxitems(FL_OBJECT *obj)
```

It is possible to add menu items to an existing menu using the call

```
int fl_addto_menu(FL_OBJECT *obj, const char *menustr)
```

The function returns the current number of menu items.

This is sometimes easier to use than defining the whole menu string at once (especially when the contents of a menu change from time to time).

Also routines exist to change a particular menu item or delete it:

```
void fl_replace_menu_item(FL_OBJECT *obj,int numb,const char *menustr)
```

```
void fl_delete_menu_item(FL_OBJECT *obj, int numb)
```

to clear the whole menu use the routine:

```
void fl_clear_menu(FL_OBJECT *obj)
```

One can change the appearance of different menu items. In particular, it is desirable to sometimes make them grey and unselectable and to put boxes with and without checkmarks in front of them. This can be done using the routine:

```
void fl_set_menu_item_mode(FL_OBJECT *obj, int numb, unsigned mode)
```

Here `mode` is the display characteristics you want to apply to the chosen entry. You can specify more than one at a time by adding or bitwise OR-ing these values together. For this parameter, the following symbolic constants exist:

`FL_PUP_NONE`	No special display characteristic. The default.
`FL_PUP_GREY`	Entry is grayed out and disabled. Not selectable.
`FL_PUP_BOX`	Entry has an empty box to the left (indicating toggle/radio.)
`FL_PUP_CHECK`	Entry has a checked box (a down box) to the left.
`FL_PUP_RADIO`	Radio entry with a box to the left.

There is also a routine that can be used to obtain the current mode of an item after interaction, mostly useful for toggle or radio items:

```
unsigned int fl_get_menu_item_mode(FL_OBJECT *ob, int numb)
```

It is often useful to define keyboard shortcuts for particular menu items. For example, it would be nice to have <ALT> `s` behave like selecting Save from a menu. This can be done using the following routine:

```
void fl_set_menu_item_shortcut(FL_OBJECT *obj,int numb,const char *str)
```

`str` contains the shortcut for the item. (Actually, it can contain more shortcuts for the same item.) See the description of the button object class for more information about shortcuts.

Finally there is the routine:

```
void fl_show_menu_symbol(FL_OBJECT *obj, int show)
```

With this routine you can indicate whether to show a menu symbol at the right of the menu label. By default no symbol is shown.

For most applications, the following routine may be easier to use at the expense of somewhat restrictive value an menu item can have. However, you can create cascade menus using this routine.

```
int fl_set_menu_entries(FL_OBJECT *ob, FL_PUP_ENTRY *ent)
```

where `ent` is a pointer to the following structure terminated by a null `text` field:

```
typedef struct
{
    const char *text;
    FL_PUP_CB callback;
    const char *shortcut;
    int mode;
} FL_PUP_ENTRY;
```

The meaning of each member is explained in Section 21.3. For menus, item callback function can be null if menu callback handles the interaction results. See demo program `popup.c` for an example use of `fl_set_menu_entries()`.

The function `fl_set_menu_entries()` works by creating and associating a popup menu with the menu object. The popup ID is returned by the function. Whenever the function is called, the old popup associated with the object, if exists, is freed and a new one created. Although you can manipulate the menu either through the menu API (but adding and removing menu items are not supported) or popup API, the application should not free the popup directly, use `fl_clear_menu()` instead.

Attributes

Any boxtype can be used for a menu except for PULLDOWN_MENU for which nobox should not be used.

Color1 controls the color of the box when not selected and color2 is the color when the menu is shown.

To change the font used in the popup menu (not the menu label), use the following routines

```
void fl_setpup_default_fontstyle(int style)

void fl_setpup_default_fontsize(int size)
```

If desired, you can attach an external popup to a menu object via the following routine

```
void fl_set_menu_popup(FL_OBJECT *ob, int pupID)
```

where pupID is the popup ID returned by fl_newpup() or fl_defpup(). If the menu type is FL_PULLDOWN_MENU, the shadow of the popup is automatically turned off. See Section 21.3 for more details on popup creation.

For a menu so created, only fl_get_menu and fl_get_menu_text work as expected. Other services such as mode query etc. should be obtained via popup routines.

To obtain the popup ID associated with a menu, use the following routine

```
int fl_get_menu_popup(FL_OBJECT *ob);
```

Function return the popup ID if the menu is created by fl_set_menu_popup() or by fl_set_menu_entries(), otherwise it returns -1.

Remarks

See menu.c for an example of the use of menus. You can also use FL_MENU_BUTTON to initiate a callback and use XPopup directly within the callback. See pup.c for an example of this approach.

19.2 Choice

Short description

A choice object is an object that allows the user the choose among a number of choices. The current choice is shown in a box. The user can either cycle through the list of choices using the left or middle mouse button or get the list as a menu using the right mouse button.

Adding an object

To add a choice object to a form use the routine

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

It shows a box on the screen with the label to the left of it and the current choice (empty in the beginning) centered in the box. The object label is also used as the title of the popup if not empty.

Types

The following types are available:

FL_NORMAL_CHOICE	Middle/right mouse button shortcut.
FL_DROPLIST_CHOICE	Menu is activated only by pressing and releasing on the arrow.

Interaction

There are two ways in which the user can pick a new choice. One way is using the right or middle mouse button. Pressing and releasing the right mouse button on the choice object sets the next choice in the list. When pressing the middle mouse button the previous choice is taken. Keeping the mouse pressed cycles through the list. The other way is to use the left mouse button. In this case a menu appears from which the user can select the proper choice. In both cases, whenever a choice is selected (even when it is the original one) the object is returned to the application program.

Other routines

There are a number of routines to change the list of possible choices. The items in the list are numbered in the order in which they are inserted. The first item has number 1, etc. Whenever the application program wants to clear the list of choices it should use the routine

```
void fl_clear_choice(FL_OBJECT *obj)
```

To add a line to a choice object use

```
int fl_addto_choice(FL_OBJECT *obj, const char *text)
```

The function returns the current item number.

To delete a line use:

```
void fl_delete_choice(FL_OBJECT *obj, int line)
```

One can also replace a line using

```
void fl_replace_choice(FL_OBJECT *obj, int line, const char *text)
```

To obtain the current choice in the choice object use the call

```
int fl_get_choice(FL_OBJECT *obj)
```

It returns the number of the current choice (0 if there is no choice). You can also obtain the actual choice text using the call

```
const char *fl_get_choice_text(FL_OBJECT *obj)
```

NULL is returned when there is no current choice.

To obtain the text of a choice item, use the following routine

```
const char *fl_get_choice_item_text(FL_OBJECT *obj, int n)
```

To obtain the total number of choices (items), use the following function

```
int fl_get_choice_maxitems(FL_OBJECT *obj)
```

One can set various attributes of an item using the following routine

```
void fl_set_choice_item_mode(FL_OBJECT *ob, int numb, unsigned mode)
```

Here *mode* is the same as that used in menu object. See also Section 21.3 for details.

You can use the follow routine to populate a choice object completely, including mode and shortcut

```
int fl_set_choice_entries(FL_OBJECT *ob, FL_PUP_ENTRY *entries)
```

where `ent` is a pointer to the following structure terminated by a null `text` field:

```
typedef struct
{
    const char *text;
    FL_PUP_CB callback;
    const char *shortcut;
    int mode;
} FL_PUP_ENTRY;
```

The meaning of each member is explained in Section 21.3 (page 195). For choice, no nested entries are permitted and the item callback functions are ignored. The function returns the number of items added to the choice object.

Finally, the application program can set the choice itself using the call

```
void fl_set_choice(FL_OBJECT *obj, int line)

void fl_set_choice_text(FL_OBJECT *obj, const char *txt)
```

where `txt` must be exactly the same as the item added in `fl_addto_choice`. For example, after the following choice is created

```
fl_addto_choice(obj,"item1 | item2 | item3");
```

You can select `item2` by using

```
fl_set_choice(obj, 2)
```

or

```
fl_set_choice_text(obj, " item2 ");
```

Note the spaces in the text.

Attributes

Don't use `FL_NO_BOX` for a choice object. Color1 controls the color of the box and color2 is the color of the text in the box.

The current choice by default is shown centered in the box. To change the alignment of the choice text in the box, use the following routine

```
void fl_set_choice_align(FL_OBJECT *ob, int align)
```

To set the font size used inside the choice object use

```
void fl_set_choice_fontsize(FL_OBJECT *obj, int size)
```

To set the font style used inside the choice object use

```
void fl_set_choice_fontstyle(FL_OBJECT *obj, int style)
```

Note that the above functions only change the font inside the choice object, not the font used in the popup. To change the font used in the popup, use the following routines

```
void fl_setpup_default_fontsize(int size)

void fl_setpup_default_fontstyle(int style)
```

See section 3.11.3 for details on font sizes and styles.

Remarks

See `choice.c` for an example of the use of choice objects.

19.3 Browser

Short description

The object class browser is probably the most powerful that currently exists in the **Forms Library**. A browser is a box that contains a number of lines of text. If the text does not fit inside the box, a scroll bar is automatically added so that the user can scroll through it. A browser can be used for building up a help facility or to give messages to the user.

It is possible to create a browser from which the user can select lines. In this way the user can make its selections from (possible) long lists of choices. Both single lines and multiple lines can be selected, depending on the type of the browser.

Adding an object

To add a browser to a form use the routine

```
FL_OBJECT *fl_add_browser(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 placed below the box by default.

Types

The following types of browsers exist (see below for more information about them):

`FL_NORMAL_BROWSER`	A browser in which no selections can be made.
`FL_SELECT_BROWSER`	In this case the user can make single line selections.
`FL_HOLD_BROWSER`	Same but the selection remains visible till the next selection.
`FL_MULTI_BROWSER`	Multiple selections can be made and remain visible till de-selected.

Hence, the difference only lies in how the selection process works.

Interaction

The user can change the position of the slider or use keyboard cursor keys (including Home, PgDn, etc.) to scroll through the text. When he/she presses the left mouse below or above the slider, the browser scrolls one page (actually one line less than a page) down or one page up. Any other mouse button scrolls one line at a time. When not using an `FL_NORMAL_BROWSER`, the user can also make selections with the mouse by pointing to the correct line or by using the cursor keys.

For FL_SELECT_BROWSER's, as long as the user keeps the mouse pressed, the current line under the mouse is highlighted. Whenever he releases the mouse the highlighting disappears and the browser is returned to the application program. The application program can now figure out which line was selected using the call fl_get_browser() to be described below. It returns the number of the last selected line. (Top line is line 1.)

An FL_HOLD_BROWSER works exactly the same except that, when the mouse is released, the selection remains highlighted.

An FL_MULTI_BROWSER allows the user to select and de-select multiple lines. Whenever he selects or de-selects a line the browser is returned to the application program that can next figure out (using fl_get_browser() described below) which line was selected. It returns the number of the last selected line. When the last line was de-selected it returns the negation of the line number. I.e., if line 10 was selected last the routine returns 10 and if line 10 was de-selected last, it returns -10. When the user presses the mouse on a non-selected line, he will select all lines he touches with his mouse until he releases it. All these lines will become highlighted. When the user starts pressing the mouse on an already selected line he de-selects lines rather than selecting them.

Other routines

There are a large number of routines to change the contents of a browser, select and de-select lines, etc.

To make a browser empty use:

```
void fl_clear_browser(FL_OBJECT *obj)
```

To add a line to a browser use

```
void fl_add_browser_line(FL_OBJECT *obj, const char *text)
```

A second way of adding a line to the browser is to use the call

```
void fl_addto_browser(FL_OBJECT *obj, const char *text)
```

The difference is that with this call the browser will be shifted such that the newly appended line is visible. This is useful when, e.g., using the browser to display messages.

Sometimes it may be more convenient to add characters to a browser without implying the starting of a newline. To this end, the following routine exists

```
void fl_addto_browser_chars(FL_OBJECT *obj, const char *text)
```

This function appends text to the last line in the browser without advancing the line counter unless text has embedded newline in it. In that case, the text before the embedded newline is appended to the last line, and the line counter is incremented. The characters after the newline, possibly with more embedded newlines in it, is then added to the browser via means similar to `fl_addto_browser()`.

You can also insert a line in front of a given line. All lines after it will be shifted. Note that the top line is numbered 1 (not 0).

```
void fl_insert_browser_line(FL_OBJECT *obj, int line,
                            const char *text)
```

Inserting into an empty browser or after the last line in the browser is the same as adding a line (`fl_add_browser_line()`).

To delete a line (shifting the following lines) use:

```
void fl_delete_browser_line(FL_OBJECT *obj, int line)
```

One can also replace a line using

```
void fl_replace_browser_line(FL_OBJECT *obj, int line,
                             const char *text)
```

Making many changes to a visible browser at the same moment, i.e., clearing it and loading it with a number of new choices, is slow because the browser is redrawn after each change. The **Forms Library** has a mechanism for avoiding this using the calls `fl_freeze_form()` and `fl_unfreeze_form()`. So a piece of code that fills in a visible browser should preferably look like the following

```
fl_freeze_form(brow->form);
fl_clear_browser(brow);
fl_add_browser_line(brow,"line 1");
fl_add_browser_line(brow,"line 2");
   . . .
fl_unfreeze_form(brow->form);
```

where `brow->form` is the form that contains object `brow`.

To obtain the contents of a particular line in the browser, use

```
const char *fl_get_browser_line(FL_OBJECT *obj, int line)
```

It returns a pointer to the particular line of text.

It is possible to load an entire file into a browser using

```
int fl_load_browser(FL_OBJECT *obj, const char *filename)
```

The routine returns whether or not the file name was successfully loaded. If the file name is an empty string the box is simply cleared. This routine is particularly useful when using the browser for a help facility. You can make different help files and load the one corresponding to the context.

The application program can select or de-select lines in the browser. To this end the following calls exist with the obvious meaning:

```
void fl_select_browser_line(FL_OBJECT *obj, int line)

void fl_deselect_browser_line(FL_OBJECT *obj, int line)

void fl_deselect_browser(FL_OBJECT *obj)
```

The last call de-selects all lines. To check whether a line is selected, use the routine

```
int fl_isselected_browser_line(FL_OBJECT *obj, int line)
```

The routine

```
int fl_get_browser_maxline(FL_OBJECT *obj)
```

returns the number of lines in the browser. For example, when the application program wants to figure out which lines in a FL_MULTI_BROWSER are selected code similar to the following can be used:

```
int total_lines = fl_get_browser_maxline(brow);
for (i=1; i <= total_lines; i++)
  if (fl_isselected_browser_line(brow,i))
     /* Handle the selected line */
```

Sometimes it is useful to know how many lines are visible in the browser. To this end, the following call can be used

```
int fl_get_browser_screenlines(FL_OBJECT *ob)
```

To obtain the last selection made by the user, e.g. when the browser is returned, the application program can use the routine

```
int fl_get_browser(FL_OBJECT *obj)
```

It returns the line number of the last selection being made (0 if no selection was made). When the last action was a de-selection (only for FL_MULTI_BROWSER's) the negative of the de-selected line number is returned.

There are also calls to influence and query top line shown in the box (i.e., influence the position of the slider).

```
void fl_set_browser_topline(FL_OBJECT *obj, int line)
int  fl_get_browser_topline(FL_OBJECT *obj);
```

Note that the topline starts from 1.

It is possible to register a callback function that gets called when a line is double-clicked. To this end, the following function can be used:

```
void fl_set_browser_dblclick_callback(FL_OBJECT *ob,
                                      void (*cb)(FL_OBJECT *,long),
                                      long data)
```

Of course, double-click callback makes most sense for `FL_HOLD_BROWSER`.

Finally there is a routine that can be used to programmatically scroll the text horizontally

```
void fl_set_browser_xoffset(FL_OBJECT *ob, FL_Coord xoff)
```

where `xoff` indicates how many pixels to scroll to the left relative to the nominal position of the text lines.

There is also a function that can be used to obtain the current xoffset if needed:

```
FL_Coord fl_get_browser_xoffset(FL_OBJECT *ob)
```

Attributes

Never use the boxtype `FL_NO_BOX` for browsers.

Color1 controls the color of the box, color2 the color of the selection. The text color is the same as the label color.

To set the font size used inside the browser use

```
void fl_set_browser_fontsize(FL_OBJECT *obj, int size)
```

To set the font style used inside the browser use

```
void fl_set_browser_fontstyle(FL_OBJECT *obj, int style)
```

See section 3.11.3 for details on font sizes and styles.

It is possible to change the appearance of individual lines in the browser. Whenever the line starts with the symbol `@` the next letter indicates the special characteristic associated with this line. The following possibilities exist at the moment:

`f`	Fixed width font.
`n`	Normal (Helvetica) font.
`t`	Times-Roman like font.
`b`	Boldface. Modifier
`i`	Italic. Modifier
`l`	Large (new size = FL_LARGE_SIZE).
`m`	Medium (new size = FL_MEDIUM_SIZE).
`s`	Small (new size = FL_SMALL_SIZE).
`L`	Large (new size = current size + 6)
`M`	Medium (new size = current size + 4)
`S`	Small (new size = current size - 2).
`c`	Centered.
`r`	Right aligned.
`_`	Draw underlined text.
`-`	An engraved separator. Text following - is ignored.
`C`	The next number indicates the color index of this line.
`N`	Non-selectable line (in selectable browsers).
`@`	Regular `@` character.

More than one option can be used by putting them next to each other. For example, `@C1@l@f@b@cTitle` will give you a red, large, bold fixed font, centered word Title. As you can see, the font change requests accumulate and the order is important, i.e., `@f@b@i` gives you a fixed bold italic font while `@b@i@f` gives you a (plain) fixed font.

One word of caution is required here: The line spacing inside the browser is not changed! Hence, when using a large line, you had better take care that there is an empty line above and below it. In some cases the character `@` might need to be placed at the beginning of the lines without introducing the special meaning mentioned above. In this case you can use `@@` or change the special character to something other than `@` using the following routine

```
void fl_set_browser_specialkey(FL_OBJECT *ob, int key)
```

To align different text fields on a line, tabs (`\t`) can be embedded in the text. See `fl_set_tabstop()` for how to set tabstops.

There are routines that can be used to turn the scrollbars on and off

```
void fl_set_browser_hscrollbar(FL_OBJECT *ob, int how)

void fl_set_browser_vscrollbar(FL_OBJECT *ob, int how)
```

The following gives the possible values and their meanings:

FL_ON	Always on.
FL_OFF	Always off.
FL_AUTO	On when needed (i.e., more lines/chars than can be shown)

`FL_AUTO` is the default.

By default, the scrollbar size is determined based on the initial size of the browser. To change the default, use the following routine

```
void fl_set_browser_scrollbarsize(FL_OBJECT *ob, int hh, int vw)
```

where `hh` is the horizontal scrollbar height and `vw` is the vertical scrollbar width. Use 0 to indicate the default.

The default scrollbar type is `THIN_SCROLLBAR`. There are two ways you can change the default. One way is to use `fl_set_defaults()` or `fl_set_scrollbar_type()` to set the application wide default (preferred); another way is to use `fl_get_object_component()` to get the object handle to the scrollbars and change the the object type forcibly. The first method is preferred because the user can override the setting via resources at time. Although the second method of changing the scrollbar type is not recommended, the object handle obtained can be useful in changing the scrollbar colors etc.

Finally there is a routine that can be used to obtain the browser size in pixels for the text area

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

where `x` and `y` are measured from the top-left corner of the form (or the smallest enclosing window). To establish the relationship between the text area (a function of scrollbar size, border with and text margin), you can compare the browser size and text area size.

Remarks

There is currently a limit of maximum 2048 bytes per line for `fl_load_browser()`.

See `demo11.c` for an example program using a `FL_NORMAL_BROWSER` to view files. `browserall.c` shows all different browsers. `browserop.c` shows the insertion and deletion of lines in a `FL_HOLD_BROWSER`.

For browser class, especially multi browsers, interaction via callback is strongly suggested.

Chapter 20

Container Objects

20.1 Folders

Short description

A tabbed folder is a special container class that is capable of holding multiple groups of objects (folders) to maximize the utilization of the screen real estate. Each folder has its own tab the user can click on to call up a specific folder from which option can be indicated (See Fig. 20.1).

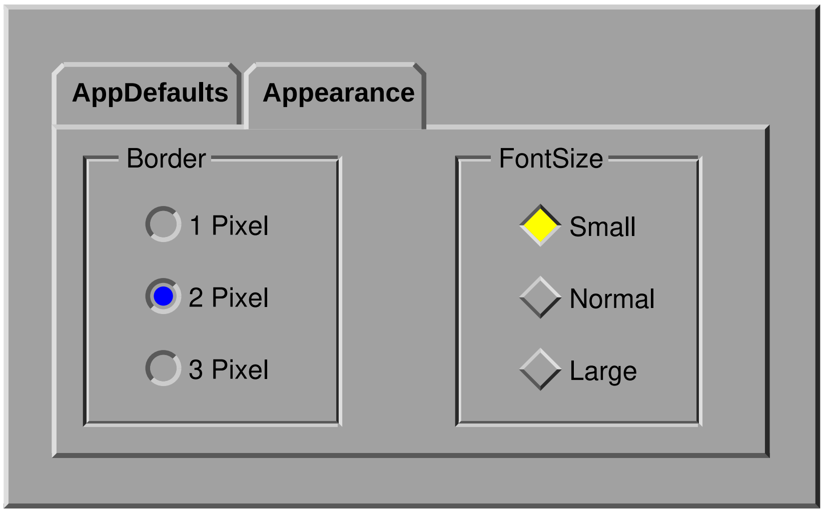

Figure 20.1: A Tabbed Folder

Adding an object

To add a tabbed folders to a form use the routine

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

```
FL_Coord w, FL_Coord h, const char *label)
```

The geometry indicated by x,y,w, and h is the total area of the tabbed folders, including the tab area.

Types

The following types are available:

FL_TOP_TABFOLDER	Tabs on top of the folders.
FL_BOTTOM_TABFOLDER	Tabs at the bottom of the folders.
FL_LEFT_TABFOLDER	Tabs on the left of the folders (not yet functional).
FL_RIGHT_TABFOLDER	Tabs on the righ of the folders (not yet functional).

Interaction

A folder as used in the tabbed folder class is simply a regular form (FL_FORM) with contents of FL_OBJECTs. Each folder is associated with a name (the tab). The folder interacts with the user just like any other form. Different from other top-level forms is that only one folder is active at any time. The user selects different folders by clicking on the tab associated with a folder. When this happens, the tab folder's callback is invoked to inform the application of this state change so the application can take appropriate actions. To find out which folder is currently active, the following routines are available

```
FL_FORM *fl_get_active_folder(FL_OBJECT *ob)

int fl_get_active_folder_number(FL_OBJECT *ob)

const char * fl_get_active_folder_name(FL_OBJECT *ob)
```

All three functions essentially perform the same task, i.e., return a handle of the active folder, but the handle returned is different. The first function returns the form associated with the folder; the second function the folder sequence number starting from 1 on the left; and the third the folder name. Depending on the application setup, one routine might be more convenient than the other two.

To find out what the last active folder is (which may be of greater interest than the currently active one) the following can be used:

```
FL_FORM *fl_get_folder(FL_OBJECT *ob)

int fl_get_folder_number(FL_OBJECT *ob)

const char *fl_get_folder_name(FL_OBJECT *ob)
```

Again, depending on the application, one might prefer one routine to the other two.

Other routines

To populate a tabbed folder, use the following routine

```
FL_OBJECT * fl_addto_tabfolder(FL_OBJECT *ob, const char *tab,
                               FL_FORM *folder)
```

where `tab` a string (with possible embedded newlines in it) indicating the title of the folder tab and `folder` is a regular form created by `fl_bgn_form()` and `fl_end_form()` pair. Only the form pointer is recorded. This means that the application program should not destroy the form that is added to the tabbed folder. The function returns the folder tab object that is of class `FL_BUTTON`. The initial object color, label color, and other attributes (gravities, for example) of the tab button is inherited from the tabbed folder object `ob` and the location and size of the tab are determined automatically. You can change the attributes of the returned object just like any other objects, but not all possibilities achieve pleasing appearance. Note that although there is no specific requirement of what the backface of the folder/form should be, a boxtype other than `FL_FLAT_BOX` or `FL_NO_BOX` may not look nice. If the backface of the form is of `FL_FLAT_BOX`, the associated tab will take on the color of the backface when activated.

One thing to note is that each tab must have its own form, i.e., you should not associate the same form with two different tabs. However, you can create more than one copies of the same form and use these.

Both the folder and tab currently do not scroll. This means that if the form size is larger than the folder area, the form is truncated. Eventually this limitation will be lifted.

Although a regular form (top-level) and a form used as a folder behave almost identically, there are some differences. In a top-level form, objects that do not have callbacks bound to them will be returned, when their states change, to the application program via `fl_do_forms()` or `fl_check_forms()`. When a form is used as a folder, those objects that do not have callbacks are ignored even when their states have changed. The reason for this behavior is that presumely that the application does not care while the changes take place and they only become relevant when the the folder is switched off and at that time the application program can decide what to do with these objects' states (Apply or Ignore for example). If immediate reaction is desired, just use callback functions for these objects.

To remove a folder, the following routine is available

```
void fl_delete_folder(FL_OBJECT *ob, FL_FORM *folder)

void fl_delete_folder_bynumber(FL_OBJECT *ob, int num)

void fl_delete_folder_byname(FL_OBJECT *ob, const char *name)
```

The application program can select which folder to show by using the following routine

```
void fl_set_folder(FL_OBJECT *obj, FL_FORM *folder)
```

```
void fl_set_folder_bynumber(FL_OBJECT *obj, int num)

void fl_set_folder_byname(FL_OBJECT *obj, const char *tab)
```

Since the area occupied by the tabbed folder contains the folder tab space, the following routine is available to obtain the actual folder size

```
void fl_get_folder_area(FL_OBJECT *obj, FL_Coord *x, FL_Coord *y,
                        FL_OBJECT *w, FL_OBJECT *h)
```

where x and y are relative to the (top-level) form the tabbed folder is on.

Remarks

Tabbed folder is a composite object consisting of a canvas and several foldertab buttons. Each indivisual form is shown inside the canvas. Folder switching is accomplished by some internal callbacks bound to the foldertab button. Should the application change the callback functions of the foldertab buttons, these new callback functions must take the responsibility of switching the active folder.

`fl_free_object(tabfolder)` does not free the individual forms that make up the tabfolder.

See demo program `folder.c` for an example use of tabbed folder class.

A nested tabfolder might not work correctly at the moment.

20.2 Menu Bar

Short description

A menubar is a collection of individual menus that typically control the top-level functions of an application. A menubar is different from individual menus in that a menu bar has a pre-determined interaction style.

Adding an object

To add a menubar to a form use the routine

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

Usually the width of the menubar is given a zero to indicate auto sizing so the menubar fills the entire width of the form it is on.

Types

The following types are available:

`FL_NORMAL_MENUBAR` The

Interaction

When the menubar appears the user can make a selection from any of the menus presented using the left mouse button. Dragging the mouse over different menus with mouse button down automatically activates the menu that is currently under the mouse.

Other routines

To set the actual menu for a menu object, use the routine

```
void fl_set_menubar(FL_OBJECT *obj, const char *menubarstr)
```

`menubarstr` describes the menubar similar in the form used by XPopups (See Section 21.3. In short, it should contain the menubar items, separated by a bar, e.g., "File|Edit|FAbout". The position and size of each menubar item is determined automatically unless special control sequences are embedded in the item label. Use character to introduces the control sequences.

The following sequences are supported:

`F` Right flush the items that come after.

`+n` Move this item right n pixels (0-9)

`-n` Move this item left n pixels (0-9)

`Sn` Reserve n pixels of space.

To set the menubar item entries, use the following routine

```
int fl_set_menubar_entry(FL_OBJECT *ob, const char *label,
 FL_PUP_ENTRY *ent)
```

where `label` is one of the labels in `fl_set_menubar()` and `FL_PUP_ENTRY` is a structure containing the actual popup items. The function returns the popup ID.

Attributes

Remarks

Menubar is not yet functional, but any comment on the API is welcome.

Chapter 21

Other objects

21.1 Timer

Short description

Timer objects can be used to make a timer that runs down toward 0.0 or runs up toward a pre-set value after which it starts blinking and returns itself to the application program. This can be used in many different ways. For example, to give a user a particular amount of time for a task, etc. Also a hidden timer object can be created. In this case the application program can take action at the moment the timer expires. For example, you can use this to show a message that remains visible until the user presses the OK button or until a particular amount of time has passed.

The precision of the timer is not very good. Don't count on anything better than, say, 0.05 seconds. Run demo `timerprec.c` for an actual accuracy measurement.

Adding an object

To add a timer to a form you use the routine

```
FL_OBJECT *fl_add_timer(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.

Types

There are at the moment three types of timers:

`FL_NORMAL_TIMER`	Visible, Shows label in box. Blinks if time expires.
`FL_VALUE_TIMER`	Visible, showing the time left or elapsed time. Blinks if time expires.
`FL_HIDDEN_TIMER`	Not visible.

Interaction

When a visible timer expires it starts blinking. The user can stop the blinking by pressing the mouse on it or by resetting the timer to 0.0. The timer object is returned to the application program or its callback called at the moment the time expires.

Other routines

To set the timer to a particular value use

```
void fl_set_timer(FL_OBJECT *obj, double delay)
```

`delay` gives the number of seconds the timer should run. Use 0.0 to reset/de-blink the timer.

To obtain the time left in the timer use

```
double fl_get_timer(FL_OBJECT *obj)
```

By default, a timer counts down toward zero and the value shown (for `VALUE_TIMER`) is the time left in the timer. You can change this default so the timer counts up and shows elapsed time

```
void fl_set_timer_countup(FL_OBJECT *obj, int yes_no)
```

A timer can be temporarily suspended (stopwatch) using the following routine

```
void fl_suspend_timer(FL_OBJECT *obj)
```

and can be resumed again by

```
void fl_resume_timer(FL_OBJECT *obj)
```

Unlike `fl_set_timer()`, a suspended timer keeps its internal state (total delay, time left etc) so when it is resumed, it starts from where it was suspended.

Finally there is a routine that allows the application program to change the way the time is presented in `VALUE_TIMER`:

```
typedef char *(FL_TIMER_FILTER)(FL_OBJECT *ob, double secs);
FL_TIMER_FILTER fl_set_timer_filter(FL_OBJECT *ob, FL_TIMER_FILTER filter)
```

The filter function is passed the timer ID and time (time left for countdown timer and elapsed time for countup timer) in seconds and should return the string representation of the time. The default filter returns the time in hour:minutes:seconds.fraction format.

Attributes

Never use FL_NO_BOX as boxtype for FL_VALUE_TIMER's.

Color1 controls the color of the timer. Color2 is the blinking color.

Remarks

Although with different API and the appearance of different interaction, the way a timer and timeout callback work is almost identical internally with one exception, that is you can deactivate a timer by deactivating the form it is on. While the form is deactivated, the timer's callback will not be called even if it has expired while the form is inactive. The interaction will resume when the form is activated.

See timer.c for the use of timers.

21.2 XYPlot

Short description

The xyplot object gives you an easy way to display a tabulated function generated on the fly or from an existing data file. An active xyplot is also available to model and/or change a function.

Adding an object

To add an xyplot object to a form use the routine

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

It shows an empty box on the screen with the label below it.

Types

The following types are available:

FL_NORMAL_XYPLOT	solid line.
FL_SQUARE_XYPLOT	solid line plus squares on data points.
FL_CIRCLE_XYPLOT	solid line plus circles on data points.
FL_FILLED_XYPLOT	the area under the curve is filled.
FL_POINTS_XYPLOT	only data points are shown.
FL_LINEPOINTS_XYPLOT	solid line plus data point.
FL_DASHED_XYPLOT	dashed line.
FL_DOTTED_XYPLOT	dotted line.
FL_DOTDASHED_XYPLOT	dash-dot-dash line.

FL_IMPULSE_XYPLOT	vertical line.
FL_ACTIVE_XYPLOT	accepts manipulations.
FL_EMPTY_XYPLOT	draws only the axes.

All xyplots display the curve auto-scaled to fit the plotting area. Although there is no limitation on the actual data, a non-monotonic increasing (or decreasing) X might be plotted incorrectly. For FL_ACTIVE_PLOT, the x data must be monotonically increasing.

The POINTS_XYPLOT and LINEPOINTS are special in that the application can change the symbol drawn on the data point.

Interaction

Only FL_ACTIVE_XYPLOT takes mouse events by default. Clicking and dragging the data points (marked with little squares) will change the data and result in the object returned to the application. By default, the reporting happens only when the mouse is released. In some situations, reporting changes as soon as they happen might be desirable, and in that case, use the following routine with when equal to FL_RETURN_CHANGED to force this behavior

```
void fl_set_xyplot_return(FL_OBJECT *ob, int when);
```

To obtain the current value of the point that has changed, use the following routine

```
void fl_get_xyplot(FL_OBJECT *ob, float *x, float *y, int *i);
```

where i is returned as the data index (starting from 0) while x,y is the actual data point. If no point is changed, i is set to -1.

To set or replace the data for an xyplot, use

```
void fl_set_xyplot_data(FL_OBJECT *obj, float *x, float *y, int n,
            const char *title, const char *xlabel, const char *ylabel)
```

Here x,y is the tabulated function, and n is the number of data points. If the xyplot being set exists already, old data will be cleared. Note that the tabulated function is copied internally so you can free or do whatever with x,y after the function returns.

You can also load a tabulated function from a file using the following routine

```
int fl_set_xyplot_file(FL_OBJECT *obj, const char *filename,
          const char *title, const char *xlabel, const char *ylabel);
```

The data file should be an ASCII file consisting of data lines. Each data line must have two columns indicating the (x,y) pair with space, tab or comma (,) separating the two columns. Lines that start with any of ! ; # are considered to be comments and are ignored. The functions returns the number of data points succesfully read or 0 if the file can't be opened.

To get a copy of the current FL_XYPLOT data, use

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

The caller must supply the space for the data.

All xyplot objects can be made aware of mouse clicks by using the following routine

```
void fl_set_xyplot_inspect(FL_OBJECT *ob, int yes);
```

Once an xyplot is in inspect mode, whenever the mouse is clicked and the mouse position is on one of the data point, the object is returned to the caller or whose callback is called. You can use `fl_get_xyplot()` to find out which point the mouse is clicked on.

Other routines

There are several routines to change the appearance of an xyplot. First of all, you can change the number of tic marks using the following routine

```
void fl_set_xyplot_xtics(FL_OBJECT *ob, int major, int minor);
void fl_set_xyplot_ytics(FL_OBJECT *ob, int major, int minor);
```

here `major` and `minor` are, respectively, the number of tic marks to be placed on the plot and divisions between major tic marks. In particular, `-1` suppresses the tic marks completely while `0` restores the default settings.

Note that the actual scaling routine may choose a value other than that requested if it decides that this would make the plot look nicer, thus `major minor` can only be taken as a hint to the scaling routine. However, in almost all cases the scaling routine will not generate a major that differs from the requested value by more than 3.

It is possible to label the major tic marks with alphanumerical characters instead of numerical values. To this end, use the following routines

```
void fl_set_xyplot_alphaxtics(FL_OBJECT *ob, const char *major,
                               const char *minor)

void fl_set_xyplot_alphaytics(FL_OBJECT *ob, const char *major,
                               const char *minor)
```

where `major` is a string specifying the labels with embedded character `|` that specifies major divisions. For example, to label a plot with `Monday Tuesday` etc, the `major` should be given `Monday|Tuesday|....` Parameter `minor` is currently unused and the minor divisions are set to 1, i.e, no divisions between major tic marks. Naturally the number of major/minor divisions set by this routine and `fl_set_xyplot_[x|y]tics()` can't be active at the same time and the one that gets used is the one that was set last.

To get a grided xyplot, use the following routine

```
void fl_set_xyplot_xgrid(FL_OBJECT *ob, int xgrid)
void fl_set_xyplot_ygrid(FL_OBJECT *ob, int ygrid)
```

where `xgrid` and `ygrid` can be one of the following

`FL_GRID_NONE`	No grid.
`FL_GRID_MAJOR`	Grid for the major divisions.
`FL_GRID_MINOR`	Grid for the major and minor divisions.

The grid line by default is drawn using a dotted line, which you can change using the following routine

```
void fl_set_xyplot_grid_linestyle(FL_OBJECT *ob, int style)
```

where `style` is the line style (`FL_SOLID, FL_DASH` etc). See Chapter 26 (page 234) for a complete list.

By default, the plotting area is automatically adjusted for tic labels and titles so that maximum area results. This can be undesirable in certain situations. To control the plotting area manually, the following routines can be used

```
void fl_set_xyplot_fixed_xaxis(FL_OBJECT *ob, const char *lm,
                               const char *rm)

void fl_set_xyplot_fixed_yaxis(FL_OBJECT *ob, const char *bm,
                               const char *tm)
```

where `lm` and `rm` specifies the right and left margin respectively and `bm` and `tm` specifies the bottom and top margins. The pixel amounts are computed using the current label font and size. Note that even for y-axis margins, the length of the string, not the height, is used as the margin, thus to leave space for one line of text, a single character (say `m`) or two narrow characters (say ii) should be used.

To restore automatic margin computation, set all margins to 0 (null).

To change the size of the symbols drawn on the data point, use the following routine

```
void fl_set_xyplot_symbolsize(FL_OBJECT *ob, int size)
```

where `size` should be given in pixels. The default is 4.

For `POINTS_XYPLOT` and `LINEPOINTS_XYPLOT` (main plot or overlay), the applicaiton program can change the symbol using the following routine

```
typedef void (*FL_XYPLOT_SYMBOL)(FL_OBJECT *, int Id,
                                 FL_POINT *p, int n, int w, int h);

FL_XYPLOT_SYMBOL fl_set_xyplot_symbol(FL_OBJECT *ob, int Id,
                                      FL_XYPLOT_SYMBOL symbol)
```