Forms Library

A Graphical User Interface Toolkit for Silicon Graphics Workstations

Mark H. Overmars

Department of Computer Science, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands

version 2.2
June 18, 1993

Preface

Constructing user interfaces for programs is normally a time consuming process. They are though very important to help the user work with the program in an easy and pleasant way. The last few years a large number of packages have appeared that help build up graphical user interfaces (so-called GUI's) in a simple way. Most of them though are difficult to use and/or expensive to buy and/or limited in their possibilities. The **Forms Library** was constructed to remedie this problem. The design goals when making the **Forms Library** were to create a package that is simple to use, powerful, graphically good looking and easily extendable.

The main notion in the **Forms Library** is that of a form. A form is a window (normally without a border) on which different objects are placed. Such a form is displayed and the user can interact with the different objects on the form to indicate his/her wishes. Many different classes of objects do exist like e.g. buttons that the user can push with the mouse (in many different flavors), sliders with which the user can indicate a particular setting, input fields in which the user can provide textual input, menu's from which the user can make choices, browsers in which the user can scroll through large amounts of text (e.g. help files), etc. Whenever the user changes the state of a particular object on one of the forms displayed the application program is notified and can take action accordingly. There are a number of different ways in which the application program can interact with the forms, ranging from very direct (waiting until something happens) to the use of call-back routines that are called whenever an object changes state.

The application program has a large amount of control over how objects are drawn on the forms. It can set color, shape, text style, text size, text color, etc. In this way forms can by tuned to ones liking.

The **Forms Library** consists of a large number of C-routines to build up interaction forms with buttons, sliders, input fields, dials, etc. in a simple way. The routines can be used both in C and in C++ programs. The library uses the Graphical Library of Silicon Graphics and should run on all Silicon Graphics machines (although some graphical power is required).

The library is easy to use. Defining a form takes a few lines of code and interaction is fully handled by the library routines. A number of demo programs are provided to indicate how easy forms are built up and used. For simple forms to e.g. ask a question or select a file name special routines are provided. E.g. to let the user choose a file in a graphical way (allowing him to run through the directory hierarchy with a few mouse clicks) the application program needs to use just one line of code.

To make designing forms even easier a **Form Designer** is provided. This is a program that lets you interactively design forms and generates the corresponding C-code. You simply choose the objects you want to place on the forms from a list and draw them on a form. Next you can set attributes, change size and position of the objects, etc., all using the mouse.

This document describes all you need to know about using the **Forms Library**. It consists of four parts.

The first part is a tutorial that describes the basics of using the library. It contains many examples of its use. This part should be read by everybody that wants to use the library.

Part two describes the **Form Designer** with which you can design forms in an interactive way.

Part three gives an overview of all object classes currently available in the library. The tutorial part only mentions the most basic classes but here you find a complete overview.

Adding new object classes to the system is not very complicated. Part four describes how this should be done.

Bugs, Extensions, etc.

Although the **Forms Library** has gone through extensive testing there are surely a number of bugs. Your comments would be greatly appreciated. Please send any bug reports to me using the Email address `markov@cs.ruu.nl`. Also if you make nice applications, new object classes, etc. please mail them to me such that I might add them to future releases.

Notice

Acknowledgments

The work on this package was partially supported by the Netherlands Organization for Scientific Research (N.W.O.). I would like to thank everybody that provided comments and suggestions after the previous versions of the **Forms Library**. Your comments have been most helpful.

Contents

Part I

Using the Forms Library

Chapter 1

Introduction

The **Forms Library** is a library of C-routines that allows you to build up interaction forms with buttons, sliders, input fields, dials, etc. in a very simple way. The **Forms Library** uses the Graphical Library of Silicon Graphics and should run on all Silicon Graphics machines (although some graphical power is required). It can be used both in C-programs and in C++ programs.

The basic way of using the **Forms Library** is as follows. First one or more forms are defined, by indicating what objects should be placed on them and where. Types of objects that can be placed on the forms include: boxes, texts, sliders, buttons, dials, input fields and many more. Even a clock can be placed on a form with one command. After the form has been defined it is displayed on the screen and control is given to a library call `fl_do_forms()`. This routine takes care of the interaction between the user and the form and returns as soon as some change occurs in the status of the form due to some user action. In this case control is returned to the program (indicating the object changed) and the program can take action accordingly, after which control is returned to the `fl_do_forms()` routine. Multiple forms can be handled simultaneously by the system and can be combined with windows of the application program. (In fact, the **Forms Library** also supports more advanced ways of interaction like e.g. the use of call-back routines.)

The **Forms Library** is simple to use. Defining a form takes a few lines of code and interaction is fully handled by the library routines. A number of demo programs are provided to indicate how easy forms are built up and used. They can be found in the directory `forms/DEMOS`. Looking at them and trying to understand them is a good way in learning the system.

If you only have very simple applications for the **Forms Library**, e.g., to ask the user for a file name, or ask him a question or give him a short message, chapter 6 contains some even more simple routines for this. So, e.g., a form with the question: *Do you want to quit* can be made with one line of code.

To make designing forms even easier a **Form Designer** is provided. This is a program that lets you interactively design forms and generates the corresponding C-code. See part two for its use.

The current version of the software is already quite extended but we are working on further improvements. In particular, we plan on designing new classes of objects that can be placed on the forms. Adding classes to the system is not very complicated. Part four of this document describes in detail how to do this yourself.

The following chapters will lead you through the different aspects of designing and using forms. In chapter 2 we give some small and easy examples of the design and use of forms. In chapter 3 we describe how to define forms. This chapter just contains the basic classes of objects that can be placed on forms. Also, for some classes only the basic types are described and not all. For an overview of all classes and types of objects see part 3 of this document. Chapter 4 describes how to set up interaction with forms. A very specific class of objects are free objects. The application program has full control over their appearance and interaction. They can be used to place anything on forms that is not supported by the standard objects. Chapter 5 describes their use. Finally chapter 6 describes some routines for simple interaction like asking questions to the user.

Chapter 2

Some examples

Before using forms for interaction with the user you first have to define them. Next you can display them and perform interaction with them. Both stages are simple. Before explaining all the details let us first look at some examples. A very simple form definition would look as follows:

```
FL_FORM *simpleform;

simpleform = fl_bgn_form(FL_UP_BOX,400.0,400.0);
  fl_add_button(FL_NORMAL_BUTTON,140.0,160.0,120.0,80.0,"Push Me");
fl_end_form();
```

The first line indicates the start of the form definition. *simpleform* will later be used to indicate the form. The type of the form is `FL_UP_BOX`. This means that the background of the form is a box that looks like coming out of the screen. The form has size 400 by 400. Next we add a button to the form. The type of the button is `FL_NORMAL_BUTTON` which will be explained below in detail. It is centered in the form by providing the bottom left corner and the width and height and has as a label *Push Me*. After having defined the form we can display it using the call

```
fl_show_form(simpleform,FL_PLACE_MOUSE,FALSE,NULL);
```

This will show the form on the screen at the mouse position. (The third argument indicates whether the form gets a border. In most cases you do not want the form to have a border so it should be FALSE as in the above example.)

Next we have to give the control over the interaction to the **Forms Library** using the call

```
fl_do_forms();
```

This will handle interaction with the form until you press the button with the mouse at which moment control is returned to the program. Now the form can be removed using

```
fl_hide_form(simpleform);
```

Note that all routines of the **Forms Library** start with `fl_`. This will avoid naming conflicts with other routines. Also, all constants start with `FL_`. The complete program is given in the file demo01.c in the directory `forms/DEMOS`. All demonstration programs can be found in this directory. Looking at them is a good way of learning how the library works.

Compile it and run it to see the effect. To compile a program using the **Forms Library** use the following arguments

```
cc -O -o demo01 demo01.c -lforms -lfm_s -lgl_s -lm
```

(Of course the library has to be installed first. Contact your systems person or read the Readme file in the directory `forms` to see how to do this.)

This simple example is of course of little use. Let us look at a slightly more complicated one (the complete program can be found in demo02.c):

```
#include "forms.h"

FL_FORM *form;
FL_OBJECT *yes, *no, *but;

fl_init();

form = fl_bgn_form(FL_UP_BOX,320.0,120.0);
  fl_add_box(FL_NO_BOX,160.0,80.0,0.0,0.0,"Do you want to Quit?");
  yes = fl_add_button(FL_NORMAL_BUTTON,40.0,20.0,80.0,30.0,"Yes");
  no  = fl_add_button(FL_NORMAL_BUTTON,200.0,20.0,80.0,30.0,"No");
fl_end_form();

fl_show_form(form,FL_PLACE_MOUSE,FALSE,NULL);
do  but = fl_do_forms(); while (but != yes);
fl_hide_form(form);
```

It creates a form with a simple text and two buttons. After displaying the form `fl_do_forms()` is called. This routine returns the object being pushed. Simply checking whether this is object *yes* determines whether we should quit.

As you see, the program starts by calling the routine `fl_init()`. This routine initializes the **Forms Library** . It should be called before any other calls to library routines are made. (Actually, if the first library routine called is `fl_bgn_form()` as in most programs, like the one above, the call to `fl_init()` can be ommitted. It is though good practise to always use it. It does not hurt and avoids problems.) One of the things the routine does is to adapt the graphical mode in which forms are being displayed to the type of machine the program is running on. This can though be changed by the application program. See the appendices for further details.

The above program can in fact be made a lot simpler, using the goodies described in chapter 6. You can simply write:

```
while (! fl_show_question("Do you want to Quit?","","") ) ;
```

It will have exactly the same effect.

Chapter 3

Defining forms

In this chapter we will describe the basics of defining forms. Not all possible classes of objects are described here, only the most important ones. Also, for most classes only a subset of the available types are described. See part three for a complete overview of all object classes currently available.

Normally you will almost never write the code to define forms yourself because the package includes a **Form Designer** that does this for you (see part two). Still it is useful to read through this chapter because it explains you what the different object classes are and how to work with them.

3.1 Starting and ending a form definition

A form consists of a collection of objects. A form definition is started with the routine

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

w and *h* indicate the width and height of the form, i.e., the largest x-coordinate that can be used in the form and the largest y-coordinate. Positions in the form will be indicated with floats between 0.0 and w or h. The actual size of the form when displayed on the screen can still be varied. *type* indicates the type of the background drawn in the form. The background is a box. See the next section for the different types available and their meaning. The routine returns a pointer to the form. This pointer must be used when e.g. drawing the form or doing interaction with it. The form definition ends with

```
void fl_end_form()
```

Between these two calls objects are added to the form. The following sections describe all the different classes of objects that can be added to a form.

Many different forms can be defined and displayed when required. It is a good habit to first define all your forms before starting the actual work.

3.2 Boxes

The first type of objects are boxes. Boxes are simply used to give the dialogue forms a nicer appearance. They can be used to visually group other object together. The bottom of each form is a box. To add a box to a form you use the routine

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

type indicates the shape of the box. The **Forms Library** at the moment supports the following types of boxes:

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 border.
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_NO_BOX	No box at all, only a centered label.

x and y indicate the lower left corner of the box in the form. w and h are the width and height of the box. *label* is a text that is placed in the center of the box. If you don't want a label in the box, use an empty string. The label can be either one line or multiple lines. To this end put newline characters (`\n`) in the label string.

The routine `fl_add_box()` returns a pointer to the box object. (All routines that add objects return a pointer to the object.) This pointer can be used for reference.

It is possible to change the appearance of a box in a form. First of all it is possible to change the color of the box and secondly it is possible to change color, size and position of the label inside the box. Details on changing attributes of objects can be found in section 3.10. Just a simple example has to suffice here. Assume we want to create a red box, coming out of the screen with the large words "I am a Box" in green in the center:

```
FL_OBJECT *thebox;

thebox = fl_add_box(FL_UP_BOX,20.0,20.0,100.0,100.0,"I am a Box");
fl_set_object_color(thebox,1,0);          /* make box red */
fl_set_object_lcol(thebox,2);             /* make label green */
fl_set_object_lsize(thebox,FL_LARGE_FONT); /* make label large */
```

Of course, this has to be placed inside a form definition.

3.3 Text

A second type of object is text. Text can be placed at any place on the form in any color you like. Placing a text object is done with the routine

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

type indicates the shape of the text. The **Forms Library** at the moment supports only one type of text: `FL_NORMAL_TEXT`.

The text can be placed inside a box using the routine `fl_set_object_boxtype()` to be described below in section 3.10. The text can be either one line or multiple lines. To this end put newline characters (`\n`) in the label string. The style of the text and the size and color can be controlled and changed in many ways. See section 3.10.

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. Text is normally placed inside the box at the left side. This helps you putting different lines of text below each other. Labels inside boxed are default centered in the box. You can change the position of the text inside the box using the routines in section 3.10. Note that, when not using any box around the text there is no need to specify a width and height of the box; they can both be 0.0.

3.4 Buttons

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. Adding a button to a form can be done using the following routine

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

label is the text placed inside (or next to) the button. *type* indicates the type of the button. The **Forms Library** at the moment supports a number of types of buttons. The most important of them are:

```
FL_NORMAL_BUTTON
FL_PUSH_BUTTON
FL_TOUCH_BUTTON
FL_RADIO_BUTTON
```

They all look the same 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 them. What actually happens when the user does so depends on the type of button. A normal button gets released when the user releases the mouse button. A push button remains pushed and is only released when the user pushes it again. A touch button is like a normal button except that, as long as the user keeps the mouse pressed it is returned to the application program (see chapter 4 on the interaction).

A 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, see below) are released. In this way the user can make its choice among some possibilities.

Whenever the user pushes a button and releases the mouse again, the interaction routine `fl_do_forms()` is interrupted and returns a pointer to the button that was pushed. In this way, the application program knows that the button was pushed and can take action accordingly, before returning control to `fl_do_forms()` again. See chapter 4 for details on the interaction with forms.

Different types of buttons are used in all the example programs provided. The application program can also set a button to be pushed or not itself without a user action. (This is of course only useful for push buttons and radio buttons. Setting a radio button does not mean that the currently set radio button is reset. The application program has to do this.) To set or reset a button use the routine

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

pushed indicated whether the button should be pushed (1) or released (0). To figure out whether a button is pushed or not use

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

See the program demo03.c for an example of the use of push buttons and setting and getting button information.

The color and label of buttons can again be changed using the routines in section 3.10.

There are two other classes of buttons available, that behave the same way as buttons but only look different.

Light buttons have a small "light" in the button. Pushing the button switches the light on and releasing the button switches it off. To add a light button use `fl_add_lightbutton()` with the same parameters as normal buttons. The other routines are exactly the same as for normal buttons. Light buttons are especially nice as radio buttons. The color of the light can be controlled with the routine `fl_set_object_color()`. See section 3.10.

Round buttons are buttons that are round. Use `fl_add_roundbutton()` to add a round button to a form. Playing with different boxtypes, colors, etc., you can make many different types of buttons. See demo04.c for some examples.

3.5 Sliders

Sliders are useful in letting the user indicate a value between some fixed bounds. A slider is added to a form using the routine

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

The two most important types of sliders are: `FL_VERT_SLIDER` displays a slider that can be moved vertically and `FL_HOR_SLIDER` gives a slider that moves horizontally. In both cases the label is placed below the slider. Default the value of the slider runs between 0.0 and 1.0. The starting position is 0.5. These values can be changed using the routines:

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

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

Whenever the value of the slider is changed by the user, the slider is returned to the application program. The program can now read its value using the call

```
float fl_get_slider_value(FL_OBJECT *obj)
```

and take action accordingly. See the example program demo05.c for the use of these routines.

3.6 Input fields

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. To add an input field to a form use

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

The main type of input field available is `FL_NORMAL_INPUT`. The input field normally looks like a `FL_DOWN_BOX`. This can be changed using the routine `fl_set_object_boxtype()` to be described in section 3.10.

Whenever the user presses the mouse inside an input field a cursor will appear in it (and it will change color). Further input will appear inside this field. The user can use the <Backspace> key to erase characters and the <Esc> key to erase the whole input field. Also the left and right arrow keys can be used to move in the field. When the user presses <Return> or <Tab> the input field is returned to the application program and input is directed to the next input field. (The <Return> key only works if there are no default buttons in the form. See the overview of object classes. The <Tab> key always works.)

The user can use the mouse to select parts of the input field which will be removed when the user types the <Backspace> key or replaced by what the user types in. Also the cursor can be located at different positions in the input field using the mouse.

The application program can also direct the focus to a particular object using the call

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

It puts the input focus in the form *form* onto object *obj*.

Note that the label is not the default text in the input field. The label is (default) placed in front of the input field. To set the contents of the input field use the routine

```
void fl_set_input(FL_OBJECT *obj,char str[])
```

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

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

See the program demo06.c for an example of the use of input fields.

3.7 Grouping objects

Objects inside a definition can be grouped together. To this end we put them in between the routines

```
FL_OBJECT *fl_bgn_group()
```

and

```
void fl_end_group()
```

Groups should never be nested. Groups are useful for two reasons. First of all it is possible to hide groups of objects (see section 3.8 below). This is often very handy. We can, for example, display part of a form only when the user asks for it.

The second reason is for using radio buttons. As indicated in section 3.4 pushing a radio button makes that the currently pushed radio button gets released. In fact, this happens only with radio buttons in the particular group. So to make two pairs of radio buttons (or more) simply put each pair in a different group. They won't interfere with each other. See e.g the example program demo04.c. It is a good habit to put radio buttons always in a group, even if you have only one set of them.

3.8 Hiding and showing objects

It is possible to temporarily hide certain objects or groups of objects. To this end, use the routine

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

obj is the object to hide or the group of objects to hide. Hidden objects don't play any role anymore. All routines on the form act as if the object does not exist. To make the object or group of objects visible again use

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

Hiding and showing (groups of) objects is useful to change the appearance of a form depending on particular information provided by the user. You can also make overlapping groups in the form and take care that always only one of them is visible.

3.9 Deactivating and reactivating objects

Sometimes you might want a particular object to be temporarily not active, e.g., you want to make it impossible for the user to press a particular button or to type input in a particular field. For this you can use the routine

```
void fl_deactivate_object(FL_OBJECT *obj)
```

obj is the object to be deactivated. When *obj* is a group the whole group is deactivated. To reactivate the group or button use the routine

```
void fl_activate_object(FL_OBJECT *obj)
```

Normally you also want to give the user a visual indication that the object is not active. This can e.g be done by changing the label color to grey. (See below.)

3.10 Changing attributes

There are a number of general routines that can be used to alter the appearance of any object.

3.10.1 Color

To change the color of a particular object use the routine

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

col1 and *col2* are indices into the colormap. Which colors are actually changed depends on the type of objects. For boxes and text only *col1* is important. It indicates the color of the box or of the box in which the text is placed. For buttons, *col1* is the color of the button when released and *col2* is the color of the button when pushed. (Note that when changing the color of a button the nice property that the color of a button changes when the mouse moves over it disappears.) For light buttons the two colors indicate the color of the light when off and when on. For sliders *col1* is the color of the background of the slider and *col2* is the color of the slider itself. Finally, for input objects *col1* is the color of the input field when it is not selected and *col2* is the color when it is selected. For all types of objects, the default colors can be found in the file forms.h. For example, for input fields the default colors are `FL_INPUT_COL1` and `FL_INPUT_COL2`.

Note that, although the color of an object is indicated as an index in the colormap, it might in fact be an RGB color. Hence, changing the colormap value for this index later won't result in a change of color for the object. The **Forms Library** uses its own internal colormap. To change the values of this internal colormap use the call `fl_mapcolor()` instead of the GL routine `mapcolor()`. Similar, to obtain the internal colormap color belonging to a particular entry, use `fl_getmcolor()`. An actual redraw of the object (see below) whose color is changed is required to have the change take effect.

3.10.2 Bounding boxes

Each object has a bounding box. This bounding box can have different shapes. For boxes it is determined by the type. For text it is normally not visible. For input fields it normally is a `FL_DOWN_BOX`, etc. The shape of the box can be changed using the routine

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

boxtype should be one of the following: `FL_UP_BOX`, `FL_DOWN_BOX`, `FL_FLAT_BOX`, `FL_BORDER_BOX`, `FL_SHADOW_BOX`, `FL_ROUNDED_BOX`, `FL_RFLAT_BOX`, `FL_RSHADOW_BOX` and `FL_NO_BOX`, with the same meaning as the type for boxes. Some care has to be taken when changing boxtypes. In particular, for objects like sliders, input fields, etc. never use the boxtype `FL_NO_BOX`. Don't change the boxtype of objects that are visible on the screen. It might have undesirable effects, or redraw the entire form after doing so (see below). See the program demo17.c for the effect of the boxtype on the different classes of objects.

3.10.3 Label attributes

There are also a number of routines to change the appearance of the label. The first one is

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

It sets the color of the label. The default here is black (0). The size of the label can be changed using the routine

```
void fl_set_object_lsize(FL_OBJECT *obj,float lsize)
```

lsize gives the size in points. Any size can be used. Three predefined sizes exist, namely `FL_SMALL_FONT`, `FL_NORMAL_FONT` and `FL_LARGE_FONT`. Default `FL_NORMAL_FONT` is used.

Labels can be drawn in many different font styles. The style of the label can be controlled with the routine

```
void fl_set_object_lstyle(FL_OBJECT *obj,int lstyle)
```

The following default styles exist:

`FL_NORMAL_STYLE`	Normal text.
`FL_BOLD_STYLE`	Boldface text.
`FL_ITALIC_STYLE`	Guess what.
`FL_FIXED_STYLE`	Fixed width (especially good for tables).

These correspond to the first 4 fonts. The package though can handle up to 32 different fonts. The first 16 (numbers 0–15) have been predefined. The following table gives their names:

0	Helvetica
1	Helvetica-Bold
2	Helvetica-Oblique
3	Courier
4	Courier-Bold
5	Courier-Oblique
6	Times-Roman
7	Times-Bold
8	Times-Italic
9	Bookman-Light
10	Bookman-Demi
11	Bookman-LightItalic
12	Symbol
13	Type
14	Iris
15	ZapfDingbats

The other 16 fonts (numbers 16–31) can be filled in by the application program. Actually, the application program can also change the first 16 fonts if required. For example, to change the default font it should change font number 0. To change a font use the following routine:

```
void fl_set_font_name(int style, char name[])
```

style is the number of the font (between 0 and 31) and *name* must contain the font name (as known to the IRIX font manager). See demo35.c for a demonstration of all the different font styles available.

You can also change the alignment of the label with respect to the bounding box of the object. For this you use the routine

```
void fl_set_object_align(FL_OBJECT *obj,int align)
```

The following possibilities exist:

`FL_ALIGN_LEFT`	To the left of the box.
`FL_ALIGN_RIGHT`	To the right of the box.
`FL_ALIGN_TOP`	Above the box.
`FL_ALIGN_BOTTOM`	Below the box.
`FL_ALIGN_CENTER`	In the middle of the box.

Normally, `FL_ALIGN_LEFT` places the object to the left of the bounding box, `FL_ALIGN_RIGHT` to the right, etc. For texts though the label is always placed inside the bounding box and e.g. `FL_ALIGN_LEFT` places the label at the left inside the box.

Finally, the routine

```
void fl_set_object_label(FL_OBJECT *obj,char label[])
```

changes the label of a given object.

3.10.4 Redrawing objects

A word of caution is required. It is possible to change the attributes of an object at any time. But when the form is already displayed on the screen some care has to be taken. Whenever changing attributes the system redraws the object. This is fine when drawing the object erases the old one but this is not always the case. For example, when placing labels outside the box (not using `FL_ALIGN_CENTER`) they are not correctly erased. It is always possible to force the system to redraw an object using

```
void fl_redraw_object(FL_OBJECT *obj)
```

When the object is a group it redraws the complete group. To redraw an entire form, use

```
void fl_redraw_form(FL_FORM *form)
```

Use of these routines is normally not necessary and should be kept to a complete minimum.

3.10.5 Changing many attributes

Whenever you change an attribute of an object in a visible form the object is redrawn immediately to make the change visible. This can be undesirable when you change a number of attributes of the same object. You only want the changed object to be drawn after the last change. Drawing it after each change will give a flickering effect on the screen. This gets even worse when you e.g. want to hide a few objects. After each object you hide the entire form is redrawn. Except for the flickering, it is also time consuming, especially when you work in double buffered mode. In this case every redraw (even of a very simple object) takes about 1/60th of a second. Changing e.g. the contents of 20 fields would take 1/3th of a second which is a noticible delay. So you would like to tell the library to temporarily not redraw the form while changes are being made. This can be done by "freezing" the form. While a form is being frozen it is not redrawn. Only when you unfreeze the form, all changes made in the meantime are drawn at once. For freezing and unfreezing two calls exist:

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

and

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

It is a good habbit to place multiple changes to the contents of a form always between calls to these two procedures. (Note that in old version of the library it was also possible to freeze individual objects. Starting with version 2.2 these calls have become obsolete and should no longer be used.)

3.11 Symbols

Rather than textual labels, it is also possible to place symbols like e.g. arrows on objects. This is done in the following way:

When the label starts with the character `@` no label is drawn but a particular symbol is drawn instead. The rest of the label string indicates the symbol. A number of predefined symbols are available. The most important ones are arrows. The following types of symbols are available at the moment:

`->`	Normal arrow pointing to the right.
`<-`	Normal arrow pointing to the left.
`>`	Triangular arrow pointing to the right.
`<`	Triangular arrow pointing to the left.
`>>`	Double triangle pointing to the right.
`<<`	Double triangle pointing to the left.
`<->`	Arrow pointing left and right.
`arrow`	A thin arrow pointing to the right.
`square`	A square.
`circle`	A circle.
`line`	A horizontal line.
`plus`	A plus sign (can be rotated to get a cross).

See demo25.c for all of them. It is possible to put the symbols in different orientations. When the symbol name is preceded by a digit 1–9 (not 5) it is rotated like on the numerical keypad. I.e., 6 indicates no rotation, 9 a rotation about 45 degrees counter clockwise, 8 a rotation of 90 degrees, etc. Hence the order is 6, 9, 8, 7, 4, 1, 2, 3. (Just think of the keypad as consisting of arrow keys.) So to get an arrow that is pointing to the left top use as a label `@7->`. To put the symbol in other orientations, put a `0` after the `@`, followed by three digits that indicate the angle (counter-clockwise). E.g. to put an arrow under an angle of 30 degrees give as a label `@0030->`.

The symbol will be scaled to fit in the bounding box. When the bounding box is not square, scaling in the x- and y-direction will be different. To avoid this, i.e., to keep the scaling the same in both directions, put a `#` behind the `@`. E.g., `@#0045->`.

The application program can also add symbols to the system which it can then use to display symbols on objects that are not provided by the **Forms Library**. To add a symbol, use the call

```
int fl_add_symbol(char name[], void (*drawit)(),int sc)
```

name is the name under which the symbol should be known (at most 15 characters), without the leading `@`. *drawit* is the drawing routine that draws the symbol. *sc* indicates whether the symbol is scalable and rotateable. If $sc = 0$, as e.g. for a bitmap, the main routine will not try to rotate and scale it. The routine *drawit* should have the form

```
void drawit(float col)
```

col is the color in which to draw the symbol. It is the label color that can be provided and changed by the application program. The routine should draw the symbol inside a unit circle around the origin. The scaling etc. to make it fit inside the object is done by the main routine. If the symbol is not scalable it should be drawn centered around the origin.

3.12 Adding and deleting objects

In some situations you might want to add an object to an existing form. This can be done using the call

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

After this call you can again add objects to the form. To stop adding objects to the form use `fl_end_form()` as before. It is possible to add objects to forms that are being displayed but better don't because not everything behaves well (e.g. strange things might happen when a group is started but not finished yet).

To delete an object from a form simply use

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

It deletes the object from the form it is currently in. The object remains available and can be added to the same or another form later using

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

Use of these calls is discouraged because some have side effects. (E.g. strange things happen when you add the same object to multiple forms.) Also watch out with deleting group objects. Not the whole group is deleted, only the object that marks its start which gives strange effects.

3.13 Freeing objects

If the application program does not need an object anymore, it can free the memory used by the object using the call

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

After this the object can no longer be used. Take care that you delete the object from the form it is in before freeing it.

To free the memory used by an entire form use the call

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

This will free the form itself and all the objects in it.

Chapter 4

Doing interaction

After having defined the forms the application program can use them to interact with the user. As a first step the program has to display the forms with which it wants the user to interact. This is done using the routine

```
long fl_show_form(FL_FORM *form,int place,int border,char name[])
```

It opens a window on the screen in which the form is shown. The place and size of the window are determined by *place*. The following possibilities exist:

`FL_PLACE_MOUSE`	The form is placed centered below the mouse.
`FL_PLACE_SIZE`	The user can control the position but the size is fixed.
`FL_PLACE_ASPECT`	Only the aspect ratio remains as in the form definition.
`FL_PLACE_CENTER`	The form is placed in the center of the screen.
`FL_PLACE_POSITION`	Place at latest position (see also below).
`FL_PLACE_FULLSCREEN`	The form is scaled to cover the full screen.
`FL_PLACE_FREE`	Position and size are completely free.

To control the position or size precisely, the application program can use the calls

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

and

```
void fl_set_form_size(FL_FORM *form,long w,long h)
```

before placing the form on the screen. (Actually the routines can also be called while the form is being displayed. It will change shape.) x, y, w and h indicate the position of the form on the screen and its size. When the position is negative the distance from the right or the top is indicated. Next the form should be placed on the screen using `FL_PLACE_POSITION`. E.g., to place a form at the top-right of the screen use e.g.

```
fl_set_form_position(form,-20,-20);
fl_show_form(form,FL_PLACE_POSITION,FALSE,NULL);
```

In the call `fl_show_form()` the argument *border* indicates whether or not a window border should be drawn around the form. For most forms you probably do not want a window boundary. A window boundary is useful to let the user iconify a form and moving it around. (When the form has no border it normally more difficult (or even impossible) to move the form. Under IRIX 4.0 the way to do this depends on the window manager. The default manager 4Dwm in the default setting allows you to move a form by holding the <Alternate> key and pressing the right mouse button inside the window. A menu will appear from which you can choose the move option.) When the window gets a border *name* indicates its name. The routine returns the GL window handle of the form. You normally never need this. If you want to change the position or size of a form later use the routines `fl_set_form_position()` or `fl_set_form_size()` described above. An alternative is the routine

```
fl_scale_form(FL_FORM *form,float xsc,float ysc)
```

Where you indicate a scaling factor in the x- and y-direction with respect to the current size. See demo43 for an example.

Multiple forms can be shown at the same moment and the system will interact will all of them simultaneously.

The graphical mode in which the form is shown depends on the type of machine. On machines with enough bitplanes double buffered RGBmode mode will be used. This gives the nicest appearence. On 8-bit Indigos single buffered RGBmode is used. On other 8-bit systems single buffered colormap mode is used. This puts some restrictions on user defined objects classes. All standard classes though work fine. It is possible to change this behaviour. See the appendices for details.

If the application program wants to stop interaction with a form and remove it, it has to use the call

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

Note that if you don't need a form anymore you can free its memory using the call `fl_free_form()` described above.

4.1 Simple interaction

Once one or more forms are shown it is time to give the control to the library to handle the interaction with the forms. There are a number of different ways of doing this. The first one is to call

```
FL_OBJECT *fl_do_forms()
```

It controls the interaction until some object in one of the forms changes state. In this case a pointer to the changed object is returned. A change occurs in the following cases:

box A box never changes state and, hence, is never returned by `fl_do_forms()`.

text Also a text never changes state.

button A button is returned when the user presses a mouse button on it and then releases the button. The change is not reported before the user releases the mouse button, except with touch buttons which are returned all the time as long as the user keeps the mouse pressed on it. (See e.g. demo07.c for the use of touch buttons.)

slider A slider is returned whenever it changes value, so whenever the user moves his mouse after having pressed the slider.

input An input field is returned when it is deactivated, i.e., the user has selected it and then selected another input field for input (e.g. by pressing the <Tab> key).

When the object is returned by `fl_do_forms()` the application program can check what the change is and take action accordingly. See all the demo programs for examples of the use. Normally, after the action is taken by the application program `fl_do_forms()` is called again to continue the interaction. Hence, most programs have the following global form:

```
/* define the forms */
/* display the forms */
while (! ready)
{
  obj = fl_do_forms();
  /* handle the change in obj */
}
```

4.2 General interaction

The above way of interaction is fine for many application programs but not for all. When the program also has to perform tasks when no user action takes place (e.g. redrawing a rotating image all the time), another call can be used:

```
FL_OBJECT *fl_check_forms()
```

In this case control is returned to the application program immediately. When a change has occurred in some object the object is returned as with `fl_do_forms()`. But when no change has occurred control is also returned but this times `NULL` is returned as an object. Programs working in this way have as a basic structure:

```
/* define the forms */
/* display the forms */
while (! ready)
{
  obj = fl_check_forms();
  if (obj != NULL)
    /* handle the change in obj */
  /* update other things */
}
```

Note that the use of a loop like this will eat up most computer time. Hence, only use it when the application program actually has to do something all the time. Otherwise use `fl_do_forms()` instead. See demo08.c for a nice example of this. (It only works well for machines with a Z-buffer.)

4.3 Dealing with multiple windows

When displaying multiple forms you might not always want to have interaction with all of them. E.g. when the user presses a quit button in a form you might want to ask a confirmation using another form. You don't want to hide the main form because of that but you also don't want the user to be able to press buttons etc. in this form. It first has to give the confirmation. So you temporarily want to deactivate the main form. This can be done using the call

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

To reactivate the form later again use

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

It is also possible to deactivate all current forms and reactivate them again. To this end use the calls:

```
void fl_deactivate_all_forms()

void fl_activate_all_forms()
```

Note that deactivation works in an additive way, i.e., when deactivating a form say 3 times it also has to be activated 3 times to become active again.

One problem remains. Mouse actions etc. are presented to a program in the form of events in an event queue. The library routines `fl_do_forms()` and `fl_check_forms()` read this queue and handle the events. When the application program itself also opens windows, these windows should receive events as well. Unfortunately, there is only one event queue. When both the application program and the library routines read events from this one queue problems occur and events are missed. Hence, the application program should NEVER read the event queue while displaying forms. To solve this problem, the package maintains a separate event queue for the user. This queue behaves in exactly the same way as the normal event queue. To access it, the application program should use replacements for the usual GL routines. Instead of using `qdevice()`, the program should use `fl_qdevice()`, with the same parameters. Also, instead of `qread()` the program should use `fl_qread()`, etc. The following is a list of all new routines:

```
fl_qdevice()
fl_unqdevice()
fl_isqueued()

fl_qtest()
fl_qread()
```

```
fl_blkqread()

fl_qenter()
fl_qreset()

fl_tie()
```

To help find out when an event has occurred, whenever `fl_do_forms()` and `fl_check_forms()` encounter an event that is not meant for them but for the application program they return a special object `FL_EVENT`. So the basis of a program with its own windows would look as follows:

```
/* define the forms */
/* open your own window(s) */
/* display the forms */
while (! ready)
{
  obj = fl_check_forms(); /* or fl_do_forms() */
  if (obj == FL_EVENT)
  {
    dev = fl_qread(&val);
    /* handle the event */
  }
  else if (obj != NULL)
    /* handle the change in obj */
  /* update other things */
}
```

In some situations you don't want to see the user events. For example, you might want to write a subroutine that pops up some form to change some setting. This routine might not want to be concerned with any redrawing of the main window, etc., but you also don't want these events to disappear. In this case you can use the routines `fl_do_only_forms()` and `fl_check_only_forms()` that will never return `FL_EVENT`. The events don't disappear. They will be returned at later calls to the normal routines `fl_do_forms()`.

4.4 Using call-back routines

It is also possible to do interaction with forms in a quite different way, using call-back routines. Actually, for larger applications this is the recommended way of setting up interaction. There are different ways in which a program can use call-back routines. The easiest way is the following:

The application program can bind a call-back routine to any object. Whenever `fl_do_forms()` or `fl_check_forms()` would return the object, the call-back routine is called instead. If no call-back routine is bound to the object it is returned. To bind a call-back routine to an object use

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

callback is the call-back routine. *argument* is an argument that is passed to the call-back routine such that it can take different action for different objects.

The call-back routine should have the form

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

obj will be the object that caused the call-back routine to be called. *argument* is the argument as above.

See program demo09.c for an example of the use of call-back routines. Note that call-back routines can be combined with normal objects. It is possible to change the call-back routine at any moment.

When dealing with multiple forms, the application program can also bind a call-back routine to an entire form. To this end it should use the routine

```
void fl_set_form_call_back(FL_FORM *form,void (*callback)())
```

Whenever `fl_do_forms()` or `fl_check_forms()` would return an object in *form* they call the routine *callback* instead, with the object as an argument. So *callback* should have the form

```
void callback(FL_OBJECT *obj)
```

With each form you can associate its own call-back routine. When the application program also has its own windows, it probably also want to know about GL event (like e.g. mouse pushes of the user in the application windows). As stated above, this can be accomplished by checking for `FL_EVENT` objects. Another way is to add an event call-back routine. This routine will be called whenever a GL event takes place. To use an event call-back routine use the call

```
void fl_set_event_call_back(void (*callback)())
```

Whenever a GL event takes place *callback* is called with the device causing the event and the value as arguments, So *callback* should have the form

```
void callback(short dev,short val)
```

A program using all of this basically has the following form:

```
void event_cb(short dev,short val)
{ /* Handles a GL-event. */ }

void form1_cb(FL_OBJECT *obj)
{ /* Handles object obj in form1. */ }

void form2_cb(FL_OBJECT *obj)
{ /* Handles object obj in form2. */ }

main()
{
```

```
  /* create form1 and form2 and display them */
  /* create your own windows */
  fl_set_event_call_back(event_cb);
  fl_set_form_call_back(form1,form1cb);
  fl_set_form_call_back(form2,form2cb);
  fl_do_forms();
}
```

The routine `fl_do_forms()` will never return in this case. See demo27.c for a program that works this way.

It is recommended that you set up your programs using call-back routines (either for the objects or for entire forms) and an event call-back routine. This makes sure that no events are missed, events are treated in the correct order, etc. Note that different event call-back routines can be written for different stages of the program and they can be switched when required. This provides a nice way for building up programs.

Chapter 5

Free objects

In some applications the standard object classes as provided by the **Forms Library** may not be enough for your task. There are three ways of solving this problem. First of all, the application program can also open its own window in which it does interaction with the user. (See chapter 4.) A second way is to add your own object classes. (See part four of this document.) This is especially useful when your new type of objects is useful in many applications.

The third way is to add free objects to your form. Free objects are objects for which the application program handles the drawing and interaction. To add a free object use the call

```
FL_OBJECT *fl_add_free(int type,float x,float y,
         float w,float h,char label[],int (*handle)())
```

type indicates the type of free object. See below for a list and their meaning. x, y, w and h are the bounding box. The label is normally not drawn unless the handle routine takes care of this. *handle* is the routine that does the redrawing and handles the interaction with the free object. The application program must supply this routine.

This routine *handle* is called by the library whenever an action has to be performed. The routine should have the form:

```
int handle(FL_OBJECT *obj,int event,float mx,float my,char key)
```

obj is the object to which the event applies. *event* indicates what has to happen to the object. See below for a list of possible events. *mx* and *my* indicate the position of the mouse (only provided with mouse related events) and *key* is the ascii value of the key typed in by the user (only for `FL_KEYBOARD` events). The routine should return whether the status of the object has changed, i.e., whether `fl_do_forms()` or `fl_check_forms()` should return this object.

The following types of events exist for which the routine must take action:

FL_DRAW The object has to be redrawn. To figure out the size of the object you can use the fields $obj \to x$, $obj \to y$, $obj \to w$ and $obj \to h$. Some other aspects might also influence the way the object has to be drawn. E.g., you might want to draw the object differently when the mouse is on top of it or when the mouse is pressed on it. The is can be figured out as follows. The field $obj \to belowmouse$ indicates whether the object is below the mouse. The field $obj \rightarrow pushed$ indicates whether the object is currently being pushed with the mouse. Finally, $obj \rightarrow focus$ indicate whether input focus is directed towards this object. When required, also the label should be drawn. This label can be found in the field $obj \rightarrow label$. The drawing should be done such that it works correctly both in RGBmode and colormap mode. The variable `fl_rgbmode` can be consulted to check whether you should draw in RGBmode or not. There are though a number of help routines that make this quite easy. In particular, the routine `fl_color()` is provided that sets the current color to a colormap index in colormap mode (i.e., behaves like the GL routine `color()`) and sets the color to the corresponding RGB color when in RGBmode. Note that some routines are provided with the package for drawing boxes and labels. They can be used to draw parts of free objects. See chapter 23 for details on drawing objects and the routines.

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

FL_LEAVE The mouse has left the bounding box. Again, normally a redraw is enough (or nothing needs to be done).

FL_MOVE A move event is sent between `FL_ENTER` and `FL_LEAVE` events when the mouse position changes on the object (in fact, it is sent all the time, also if the mouse position remains the same). The mouse position is given with the routine.

FL_PUSH The user has pushed a mouse button in the object. Normally this requires some actual action.

FL_RELEASE The user has released the mouse button again. This event is only sent when before a PUSH event was sent.

FL_MOUSE The mouse position has changed. This event is sent to an object between a `FL_PUSH` and a `FL_RELEASE` event. The mouse position is given with the routine and action can be taken. For example, sliders use this event while e.g. buttons do not.

FL_FOCUS Input got focussed to this object. This event and the next two are only send to a free object of type `FL_INPUT_FREE` (see below).

FL_UNFOCUS Input is no longer focussed on this object.

FL_KEYBOARD A key was pressed. The ascii value is given with the routine. This event only happens between `FL_FOCUS` and `FL_UNFOCUS` events. 1 and 2 indicate the left and right arrow keys.

FL_STEP A step event is sent all the time (at most 100 times per second but often less because of time consuming redraw operations) to a free object of type `FL_CONTINUOUS_FREE` such that it can update its state or appearance.

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

As indicated not all events are sent to all free objects. It depends on their type. The following types exist:

FL_NORMAL_FREE The object will receive the events `FL_DRAW`, `FL_ENTER`, `FL_LEAVE`, `FL_MOVE`, `FL_PUSH`, `FL_RELEASE` and `FL_MOUSE`.

FL_SLEEPING_FREE The object only receives `FL_DRAW` events. This should be used for objects without interaction (e.g. a picture).

FL_INPUT_FREE Same as `FL_NORMAL_FREE` but the object also receives `FL_FOCUS`, `FL_UNFOCUS` and `FL_KEYBOARD` events.

FL_CONTINUOUS_FREE Same as `FL_NORMAL_FREE` but the object also receives `FL_STEP` events. This should be used for objects that change themselves continuously.

FL_ALL_FREE The object receives all types of event.

See demo15.c for a terrible example of the use of free objects. Free objects provide all the generality you want from the **Forms Library**. Because free object behave a lot like new object classes it is recommended that you also read part IV of this documentation before designing free objects.

Chapter 6

Goodies

A number of special routines are provided that makes working with simple forms even simpler. All these routines build up simple forms and do the interaction with the user.

6.1 Messages and questions

The following routines are meant to give messages to the user and to ask simple questions:

```
void fl_show_message(char str1[],char str2[],char str3[])
```

It shows a simple form with a message and a button with OK on it. *srt1*, *str2* and *str3* are the three lines of the message. (Use empty strings if you don't need all the lines.) The routines returns when the user presses the OK button or presses the <Return> key.

```
int fl_show_question(char str1[],char str2[],char str3[])
```

Again shows a three line message but this time with a Yes and a No button. It returns whether the user pushed the Yes button. The user can also press the <y> key to mean Yes and the <n> key to mean No.

```
int fl_show_choice(char str1[],char str2[],char str3[],
         int numb,char but1[],char but2[],char but3[])
```

Shows a three line message with one, two or three buttons. *numb* indicates the number of buttons. *but1*, *but2* and *but3* are the labels of the buttons. The routine returns the number of the button pressed (1, 2 or 3). The user can also press the <1>, <2> or <3> ket to indicate the first, second, or third button.

```
char *fl_show_input(char str1[],char defstr[])
```

This show a box with one line of message (indicated by *str1*) and an input field in which the user can enter a string. *defstr* is the default string placed in the input box. The routine returns the string when the user presses the OK button or presses the <Return> key. This routine can be used to have the user provide all kinds of textual input.

The example program demo16.c shows you these goodies.

6.2 Colormap

In a number of application the user has to select a color from the colormap. For this a goody has been created. It shows the first 64 entries of the colormap. The user can scroll through the colormap to see more entries. At the moment the user presses the mouse on some entry the corresponding index is returned and the colormap is removed from the screen. To display the colormap use the routine

```
int fl_show_colormap(int oldcol)
```

oldcol should be the current or default color. The user can decided not to change this color by pressing the cancel button in the form. The procedure returns the index of the color selected (or the index of the old color).

6.3 Buttonbox

It is also possible to create a soft buttonbox with one call. This buttonbox can be used for those people that don't have the hardware buttonbox. Whenever the user presses a button on the buttonbox, an event is placed in the event queue in the same way the normal buttonbox would do. The application program can next react to these and does not have to know whether the buttonbox is software or hardware. Of course, like normally, you have to queue the buttons on the buttonbox. For this use `fl_qdevice()`, as explained in chapter 4.

To show the buttonbox use the call

```
long fl_show_buttonbox()
```

(It returns the window handle of the buttonbox.) To hide is use

```
void fl_hide_buttonbox()
```

To set the label on one of the buttons use

```
void fl_set_buttonbox_label(int but,char label[])
```

but is the number of the button (in the range 0–31 or SW0–SW31). *label* is the label to be placed on the button (should be short).

6.4 File selector

The most extended predefined form is the file selector. It provides an easy, interactive way to let the user select files. It is called as follows:

```
char *fl_show_file_selector(char message[],char directory[],
         char pattern[],char default[])
```

A form will be shown in which all files are listed in directory *directory* that satisfy the pattern. *pattern* can be any kind of regular expression, e.g. `[a-f]*.c` which gives all files starting with a letter between a and f and ending with .c. *default* is the default file name. *message* is the message string placed at the top of the form. Now the user can choose a file from the list given.

The user can also walk through the directory structure, either by changing the directory string by pressing the mouse on it or by pressing his mouse on a directory name (shown with a D in front of it) to enter this directory.

The user can also change the pattern by pushing the mouse on it. Note that directories are shown independent of whether they satisfy the pattern. He can also type in a file name directly.

When the user is satisfied, i.e., found the correct directory and indicated the file name required he can press the button labeled Ready or press the <Return> key. He can also double click on the file name in the browser. The full path to the filename is returned by the procedure. If the user presses the Cancel button NULL is returned.

When the arguments *directory*, *pattern* or *default* are empty, the previous value is used. (Some good defaults when this happens the first time.) The application program can figure out the directory, pattern and file name (without the path) after the user changed them using the routines

```
char *fl_get_directory()
char *fl_get_pattern()
char *fl_get_filename()
```

See demo31.c for the use of the file selector.

Part II

The Form Designer

Chapter 7

Introduction

This part of the documentation describes the **Form Designer**, a program meant to interactively design dialogue forms for use with the **Forms Library**. This part assumes the reader is familiar with the **Forms Library** and has read part one of this document.

Even though designing forms is quite easy and requires only a relatively small number of lines of C-code, it is difficult to figure out the required positions and sizes of the objects. The **Form Designer** was written to facilitate the construction of forms by allowing an interactive way of placing object on forms, moving them around, scaling them, setting attributes, etc. This all works in a simple way. The **Form Designer** generates a piece of C-code that can then be included in the application program. This piece of code will contain one procedure `create_the_forms()` that the application program should call to generate all the forms designed. The code produced is easily readable.

The **Form Designer** also lets the user give names to objects for reference in the application program and allows him to bind call-back routines to objects (that have to be defined in the application program). The **Form Designer** is very simple to use. All actions are performed with the mouse or the function keys. It uses a large number of forms itself to let the user make choices, set attributes etc. Most of these forms were designed using the **Form Designer** itself.

It is important to note that the **Form Designer** only helps you in designing the layout of your forms. It does not allow you to specify the actions that have to be taken when e.g. a button is pushed. You can indicate the callback routine to call but the application program has to supply this callback routine. Also, the **Form Designer** does not allow you to initialize your objects. E.g., you cannot set the bounds of a slider inside the **Form Designer**. The bounds are not part of the layout of the form but are part of its use. Hence, it is the responsibility of the application program to initialize the objects.

Chapter 8

Getting started

To start up the **Form Designer** simply type `fdesign` without any argument. (If nothing happens, check whether the package has been installed correctly.) You will have to draw a window on the screen. This is the window in which you can create your forms. Make it large enough such that the biggest form you need will fit in it. Next the control panel appears on the screen. The main window will be black. No form is shown yet.

The control panel consists of five parts. At the left there is a list of forms you are working on. The list is empty, indicating that there are no forms yet. You can work on up to 50 forms at the same moment. You can use this list to switch from form to form. To the right of that there is a list of all groups in the form you are working on. It will be empty because there are no groups. Ignore this at the moment. For more information on the use of groups, see below. Next to this you find a list of all different types of objects that can be placed on the forms. You can use the mouse to select the type of object you want to add to the form. At the right you find a number of buttons to give commands to the program. The button at the bottom, labeled **Exit** ends the execution of the program. Other buttons will be explained below. Below this there is a row of buttons, which correspond to the function keys. You can either press the buttons with the mouse or press the function keys on the keyboard. This will have the same effect. The functions of these keys will be described below.

To create a new form click with the mouse on the button labeled **Add Form** below the (empty) list of forms. In the main window a red outline will appear. Move the mouse inside the window to change the size of this box. It indicates the size of the form you are creating (although you can change it at a later stage). Click anywhere in the window. A little notifier will appear prompting you for the name of the form. This is the name under which the application program will know the form. You will have to provide a name (which must be a legal C variable name). Type in the name and press <Return>. Now the background of the form appears in the window. Note the form name is added in the list of forms in the control panel.

To add an object to the form, choose the type of object in the control panel by clicking in the list of object classes. Next move the mouse into the form you are creating. Hold down the left <Shift> key while pressing the left mouse button. Keep the mouse button pressed and create a box that has the required size. Release the button and the object will appear. (Pressing the <Shift> key is a safeguard against unintentionally adding objects.) Note that a red outline appears around the new object. This means that the object is selected. In this way you can put all kinds of objects on the form.

It is possible to move objects around or change their size. To this end, first select the object by pressing the right mouse button in it. A red outline will appear around the object. Now, using the middle mouse button you can move the object around. By grabbing the object at one of the four red corners you can scale it. In this way you can change the layout of the objects on the form. It is also possible to select multiple objects an move or scale them simultaneously. See below for details.

To change the attributes, like e.g. the label, of an object, press the right mouse button inside the object to select it. Next, press function key <F1> (either on the keyboard or in the control form). A form will appear in which you can indicate all the different attributes. Their meaning should be clear (if you have read the documentation on the **Forms Library**). Change the attributes by pressing the right mouse button on them. A menu will appear in which you can make the required choice. Change the attributes you want to change and press the button labeled **Ready**. As you will see the changes are immediately visible. See below for more information about changing attributes.

In this way you can create the forms you want to have. Note that you can work on different forms at the same moment. Just add another form in the way described above and use the list of forms to switch between them. After you have created all your forms press the button labeled **Save forms**. It will ask you for a file name using the file selector. In this file selector you can walk through the directory tree to locate the place where you want to save the file. Next, you can type in the name of the file (or point to it when you want to overwrite an existing file). The name should end with `.fd`. So for example, choose `ttt.fd`. The program now creates three files: `ttt.c`, `ttt.h` and `ttt.fd`. `ttt.c` contains a readable piece of C code that creates the forms you designed. The file `ttt.h` contains the corresponding header file for inclusion in your application program. The file `ttt.fd` contains a description of the forms in such a way the the **Form Designer** can read it back in later. The application program now simply has to call the routine `create_the_forms()` to create the different forms you designed.

These are the basic ideas behind the **Form Designer**. Below we describe the program in detail.

Chapter 9

Command line arguments

To start the **Form Designer** simply type

```
fdesign
```

You will have to put a window on the screen in which you want to design your forms. Better make this window large enough such that the largest form you want to design fits in it. Next the program places the control panel in the right bottom corner of the screen. You can move this panel, if required, to the place you want.

When starting the **Form Designer** it is possible to specify the height and width of the main window with the options `-h` and `-w`. E.g. to make a main window of size 800 by 500 use

```
fdesign -w 800 -h 500
```

It is also possible to load one or more files with form descriptions when starting the program by placing their names (without the extension, see below) on the command line. E.g. to load the forms in the files `ttt` and `ttt2` use

```
fdesign ttt ttt2
```

or

```
fdesign -w 800 -h 500 ttt ttt2
```

Except for the main window, the **Form Designer** places its windows at fixed positions on the screen. To avoid using up too much screen space, they are drawn without borders. This makes it a bit difficult to move them. Under IRIX 4.0, (almost) all window managers still have a mechanism to move windows around. For 4Dwm, the default window manager, this works as follows. Press the right mouse button inside the window while holding down the <Alternate> key. A menu will appear. From this choose the option "move" and move the window to its new position.

In some situations it might be required to convert a `.fd` file into the corresponding `.c` and `.h` files (e.g. if you made a change by hand (not recommended) or corrupted your files in some way. Rather than starting fdesign, loading the forms and next saving them again you can use the option `-c` which takes one file name as an argument and converts it. E.g use

```
fdesign -c ttt
```

Chapter 10

Creating Forms

10.1 Creating, changing and deleting forms

To create a new form press the button labeled **Add 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.

To change the size of a form at a later stage, select the bottom box of the form, using the right mouse button. Next grab the box using the middle mouse at the top 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 **Form Name** 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 at the left of the control panel. Next hold down the left <Shift> key and draw a bounding box with the left mouse button. 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. The right mouse button is used for selecting objects. Simply press it inside the object you want to select. A red outline will appear, indicating that the object is selected.

It is also possible to select multiple objects. To this end again use the right mouse button to draw a box 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 about 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, again using the middle mouse button.

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 putting a number of equal objects on the form: First put one on the form, change the attributes as required and next copy it.

10.5 Aligning objects

Sometimes you have a number of object 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 labelled **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 **Hide** on the form.

Now select the objects you want to align. Next, press one of the alignment buttons in the form. The buttons mean: top row: align left, horizontal center, align right, horizontal equal distance (see below), bottom row: align bottoms, vertical center, align tops, vertical equal distance. 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.

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 the snapping is 10 pixels. Snapping helps in making objects the same size and putting them nicely aligned.

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 which 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.

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>. 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. 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 thecolors, push the mouse on it. A box will appear showing 64 colors in the 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 Object names and call-back routines

Three more fields can be filled in in the attributes form: name, call-back 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 such that the program can refer to it. Take care that all object names used are different. It should be legal C variable names. It is possible to use arrays of objects. E.g. if you define some objects as `tt[0]`, `tt[1]` and `tt[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.)

Call-back indicates the call-back 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. Of course, the application program will have to provide the call-back routine.

Note that when copying objects also these fields are copied. This might lead to multiple objects with the same name. This will lead to undesired effects. So be careful always to change the names after copying an object.

10.9 Cut, Copy and Paste

You can remove objects from the form by first selecting them and next pressing function key <F12>. 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.

10.10 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 remove 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. This can though be achieved very simple. 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 ungroup 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 button **Group Name** under the list of groups to change the name of a selected group. If multiple groups are selected only the name of the first group is changed.

10.11 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. Better though first group the objects before hiding them. Now you can select them by pressing the mouse on the group name in the group browser.

10.12 Testing forms

To test the current form, press the button labeled **Test Form**. 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 play with the form as long as you want. When ready, press the button **Stop Testing**.

Note that any changes you make 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 press the button labeled **Save forms**. You will be prompted for a file name using the file selector. 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`. `ttt.c` contains a piece of C-code that builds up the forms. For each form there is a separate routine that constructs it. Moreover the global creation routine is made that creates all of them. Default the name of this routine is `create_the_forms()` but it can be changed (see below). The application program should simply call this routine to create all the forms. `ttt.h` contains all the object and form names as indicated by the user. It also contains the names of the defined call-back routines. Normally the application program would look something like this:

```
#include "forms.h"
#include "ttt.h"

/* The call back routines */

main()
{
  create_the_forms();
  /* rest of the program */
}
```

Note that the file `ttt.c` is easily readable. Hence, you can always change it by hand later. This is though discouraged because you will have to redo the changes whenever you call fdesign again. Better compile it separately and never change it!

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 better not change it because not much error checking is done when reading it in. To load such a file press the button labeled **Load forms**. 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 press the button labeled **Merge forms**.

In some situations it is useful to change the name of the main creation routine to something else than `create_the_forms()`. For example in cases where you want to use forms from different sets. In this case you would want to have a different name for the creation routine in each set to avoid naming conflicts. To do so press the button labeled **Main Routine** and indicate the new name.

Part III

An overview of all object classes

Chapter 12

Introduction

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

Currently the following object classes are available

Name	Description
	Static Objects
Box	Rectangular areas to visually group objects.
Text	Simple one line labels.
Bitmap	Displays an X-style bitmap.
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.
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.
	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.

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, float x, float y,
                         float w, float h, 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, float x, float y,
                            float w, float h, 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 sections to follow.

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.

Most object classes also contain some other routines to e.g. set the value of the object or get the value when the user has interacted with the object. These are described next.

Finally it is describes how certain attributes of the objects can be changed. 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 13

Static objects

13.1 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, float x, float y,
                             float w, float h, 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_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

Boxes are used in most demo's.

13.2 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 *add_text(int type, float x, float y,
                             float w, float h, char label[])
```

The meaning of the parameters is as usual. The label is default placed left in the bounding box.

Types

Only one type of text exists: `FL_NORMAL_TEXT`.

Interaction

No interaction takes place with text objects.

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.

Remarks

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. 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_align()`. In contrast to boxes, when using different alignments, the label is always placed inside the box rather than outside the box.

13.3 Bitmap

Short description

A bitmap is a simple bitmap. The bitmaps should look like the X-bitmaps (that can be created with the program bitmap).

Adding an object

To add a bitmap to a form you use the routine

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

The meaning of the parameters is as usual. The label is 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. (You can place a hidden button over it if you want something to happen when pressing the mouse on it.)

Other routines

To set the actual bitmap being displayed use

```
void fl_set_bitmap(FL_OBJECT *ob,int width,int height,char *bits)
```

bits contains the bits in the usual way for X-bitmaps, i.e. from top to bottom. A number of bitmaps can be found in `/usr/include/X11/bitmaps`. The X program `bitmap` can be used to create bitmaps.

Attributes

Color1 controls the foreground color of the bitmap, color2 the background color (and the color of the box if the boxtype is changed).

Remarks

See demo33.c for a demo of a bitmap.

13.4 Clock

This is an adoption of the clock program by Paul Haeberli.

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, float x, float y,
                              float w, float h, char label[])
```

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

Types

The following types are available:

`FL_SQUARE_CLOCK`	A square clock.
`FL_ROUND_CLOCK`	A round clock.

Interaction

No interaction takes place with clocks.

Other routines

To get the current time use the routine

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

Attributes

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

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

Remarks

See demo12.c for an example of the use of clocks.

13.5 Chart

Short description

The chart object gives you an easy way to display a number of different types of charts like e.g. bar-charts, pie-charts, line-charts, etc. They can either be use 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, float x, float y,
                             float w, float h, 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-chart 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 (the default) `FL_CHART_MAX` which is 128. Switching between types can be done without any problem.

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, float val, char text[], int col)
```

Here *val* is the value of the item, *text* is the label to be placed with the item (can be empty) and *col* is an index in the colormap that is the color of this item. The chart will be redrawn each time you add an item. This is not what you want if you fill a chart with values. In this case put the calls between `fl_freeze_form()` and `fl_unfreeze_form()`.

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

```
void fl_insert_chart_value(FL_OBJECT *obj, int index,
                           float val, 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,
                            float val, char text[], int col)
```

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

Normally, bar-charts and line-chart 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, float min, float 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 demo37.c and demo38.c for examples of the use of chart objects.

Chapter 14

Button like objects

14.1 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.

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.

Adding an object

To add buttons use one of the following routines:

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

FL_OBJECT *fl_add_lightbutton(int type, float x, float y,
                         float w, float h, char label[])

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

The meaning of the parameters is as usual. The label is default placed inside the button, except for round buttons where it is placed to the right of the circle.

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_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 the same 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 button. A 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.

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 indicated whether the button should be pushed (1) or released (0). Note that when setting a FL_RADIO_BUTTON to be pushed this does not automatically set the currently pushed button to be released. The application program will have to do that itself. To figure out whether a button is pushed or not use

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

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 use the routine

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

It returns the number of the mouse button (1 = right, 2 = middle, 3 = left).

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, char str[])
```

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 as 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. Similar you can use the symbol `#` to indicate the <Alt> key.) Be careful with you 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. Also note that the arrow keys correspond to <Cntrl> A,B,C,D. Radio buttons cannot have shortcuts associated with them. 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.

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.

Remarks

See all demo programs, in particular demo03.c and demo04.c, for the use of buttons. `FL_INOUT_BUTTON`s are e.g. used in the buttonbox.

Chapter 15

Valuator objects

15.1 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, float x, float y,
                             float w, float h, char label[])
```

or

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

The meaning of the parameters is as usual. The label is 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.

Interaction

Whenever the user changes the value of the slider using the mouse, the slider is returned 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.

In some applications you only want the slider 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_slider_return(FL_OBJECT *obj, int always)
```

Set *always* to FALSE to achieve this goal.

Other routines

To change the value of the slider use the routines

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

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

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 slider use

```
float fl_get_slider_value(FL_OBJECT *obj)

void fl_get_slider_bounds(FL_OBJECT *obj, float *min, float *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, float step)
```

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

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, float size)
```

size should be a float between 0.0 and 1.0. When 1.0 the slider covers the box completely and can no longer be moved. The default is `FL_SLIDER_WIDTH` = 0.08. This only applies to `FL_VERT_SLIDER` and `FL_HOR_SLIDER`.

Finally, 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.

Remarks

See the demo program demo05.c for an example of the use of sliders. See demos demo23.c and demo24.c for the effect of setting slider sizes and the different types of sliders.

15.2 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, float x, float y,
                             float w, float h, char label[])
```

The meaning of the parameters is as usual. The label is 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.

Interaction

Whenever the user changes the value of the dial using the mouse, the dial is returned by the interaction routines.

In some applications you only want the dial 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_dial_return(FL_OBJECT *obj, int always)
```

Set *always* to FALSE to achieve this goal.

Other routines

To change the value of the dial use

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

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

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

```
float fl_get_dial_value(FL_OBJECT *obj)
```

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

In a number of situations you would like 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, float step)
```

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

Attributes

You can use any boxtype you like. Default is `FL_NO_BOX`.

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

Remarks

See the demo program demo10.c for an example of the use of dials.

15.3 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-value.

Adding an object

A positioner can be added to a form using the call

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

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

Types

Only one type of positioner exists at the moment:

`FL_NORMAL_POSITIONER` Cross-hair inside a box.

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 FALSE 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, float val)

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

void fl_set_positioner_yvalue(FL_OBJECT *obj, float val)

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

Default the minimum values are 0.0, the maximum values are 1.0 and the values are 0.5. To obtain the current values of the positioner use

```
float fl_get_positioner_xvalue(FL_OBJECT *obj)

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

float fl_get_positioner_yvalue(FL_OBJECT *obj)

void fl_get_positioner_ybounds(FL_OBJECT *obj, float *min, float *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_positoiner_xstep(FL_OBJECT *obj, float step)

void fl_set_positoiner_ystep(FL_OBJECT *obj, float step)
```

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

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 demo21.c.

15.4 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, float x, float y,
                             float w, float h, char label[])
```

The meaning of the parameters is as usual. The label is 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.

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 achieve this call the routine

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

Set *always* to TRUE to achieve this goal.

Other routines

To change the value of the counter use the routines

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

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

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

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

To obtain the current value of the counter use

```
float fl_get_counter_value(FL_OBJECT *obj)
```

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

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

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

See demo36.c for an example of the use of counters.

Chapter 16

Input objects

16.1 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, float x, float y,
                                  float w, float h, char label[])
```

The meaning of the parameters is as usual. The label is 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_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 else it is not shown and the bell is sounded. Similar, an int input field can only contain an integer number. A multiline 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 typing text. This can be used for e.g. passwords. Finally, a hidden input field is not shown at all but does collect text for the application program to use.

Note that input fields do not scroll scroll (neither horizontal nor vertical). So the area reserved for the field should be large enough to contain the entire text. Also note that in the current version, for each keypress the input field is redrawn completely. For multiline input fields this looks rather bad when in single buffer mode. So when using multiline input fields better force double buffer mode.

Interaction

Whenever the user presses the mouse inside an input field a cursor will appear in it (and it will change color). Further input will appear inside this field. The user can use the <Backspace> key to erase characters in front of the cursor, the <Delete> key to erase characters behind the cursor and the <Esc> key to erase the whole input field. Also the left and right arrow keys can be used to move in the field. For a multiline input field also the up and down arrows can be used. The return key will start a new line in the usual way.

The user can use the mouse to select part of the input field which will be 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.

When the user presses the <Tab> key the input field is returned to the application program and input 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 some return button. This does not work for multiline input fields. Also when the user picks a new input field with the mouse, the current input object is returned. In some cases it is useful to have the input field being returned to the application program after each keystroke of the user. To achieve this use the routine:

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

Set *always* to TRUE to achieve this goal.

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, char str[])
```

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

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

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.

Remarks

See the program demo06.c for an example of the use of input fields. See demo39.c for multiline input fields. See demo41.c for secret input fields.

Chapter 17

Choice objects

17.1 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) the pop-up menu appears. The user can then make a selection from the menu. Unfortunately, due to the way the Graphical Library works, selections from menus have to be made with the right mouse button. So even when the user pops up the menu with the left mouse button he has to select an item with the right mouse button.

Adding an object

To add a menu to a form use the routine

```
FL_OBJECT *fl_add_menu(int type, float x, float y,
                             float w, float h, 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_TOUCH_MENU	The menu appears when the user move the mouse inside it.

Interaction

When the menu appears the user can make a selection using the right 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, char menustr[])
```

menustr describes the menu in the form used by the Graphical Library. In short, it should contain the menu items, separated by a bar, e.g., “First|Second|Third”. 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.

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

```
void fl_addto_menu(FL_OBJECT *obj, char menustr[])
```

This is sometimes easier to use than defining the whole menu string at once (especially when the contents of a menu changes 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, 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 possible 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, long mode)
```

Here *mode* is the mode as used in the GL routine `setpup()`. See the manual page for that routine for details.

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, 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. Default no symbol is shown.

Attributes

Any boxtype can be used for a menu.

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

Remarks

See demo13.c for an example of the use of menus.

17.2 Choice

This class was written together with Trevor Paquette.

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, float x, float y,
                              float w, float h, 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.

Types

Only the type `FL_NORMAL_CHOICE` does exist.

Interaction

There are two ways in which the user can pick a new choice. One way is using the left or middle mouse button. Pressing and releasing the left mouse button on the choice object makes that the next choice in the list is taken. 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 right mouse button. In this case a menu does appear 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

```
void fl_addto_choice(FL_OBJECT *obj, char text[])
```

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, 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

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

NULL is returned when there is no current choice.
Finally, the application program can set the choice itself using the call

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

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.
To set the font size used inside the choice object use

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

To set the font style used inside the choice object use

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

See section 3.10.3 for details on font sizes and styles.

Remarks

See demo32.c for an example of the use of choice objects.

17.3 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 not the whole text fits inside the box, automatically a scroll bar is added such that the user can scroll through it. In this sense, the browser can be used for e.g. building up a help facility or to give messages to the user.

It is also possible to create a browser in 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, float x, float y,
                             float w, float h, char label[])
```

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

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 deselected.

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

Interaction

The user can change the position of the slider to scroll through the text. When he/she presses the mouse below or above the slider, the browser scrolls one page down or one page up. When not using a `FL_NORMAL_BROWSER`, the user can also make selections with the mouse by pointing to the correct line.

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.)

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

A `FL_MULTI_BROWSER` allows the user to select and deselect multiple lines. Whenever he selects or deselects 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 deselected 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 deselected 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 deselects lines rather than selecting them.

Other routines

There are a large number of routines to change the contents of a browser, select and deselect 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, char text[])
```

A second way of adding a line to the browser uses the call

```
void fl_addto_browser(FL_OBJECT *obj, char text[])
```

The difference is that with this call the browser will be shifted such that the new added line is visible. This is useful when e.g. using the browser for displaying messages.

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, char text[])
```

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, 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 very 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

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

where *form* is the form that contains object *brow*.

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

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

It returns a pointer to the particular line of text.

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

```
int fl_load_browser(FL_OBJECT *obj, char filename[])
```

The routine returns whether 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 users wishes.

The application program can select or deselect 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 deselects all lines. To check whether a line is selected, 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 the following piece of code can be used:

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

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 deselection (only for `FL_MULTI_BROWSER`'s) a negative line number is returned.

Finally there is a call to influence the top line shown in the box (i.e., influence the position of the slider).

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

Attributes

Never use the boxtype `FL_NO_BOX` for browsers when they change or when the text does not fit in the box.

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, float size)
```

To set the font style used inside the browser use

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

See section 3.10.3 for details on font sizes and styles.

It is possible to change the appearence of individual lines in the browser. Whenever the line starts with the symbol `@` the next letter indicates the way the line should be drawn. The following possibilities exist at the moment:

- `b` Boldface.
- `i` Italic.
- `f` Fixed width font.
- `l` Large.
- `s` Small.
- `c` Centered.
- `r` Right aligned.
- `_` A seperating line is put below the item.
- `C` The next number indicates the color index of this line.

More that one option can be used by putting them next to each other. E.g. `@C1@l@b@cTitle` will give you a red, large, bold, centered word Title. One word of caution is required here: The line spacing inside the browser is not changed! Hence, when using a large line, better take care that there is an empty line above and below it. In some cases the character `@` might appear at the beginning of the lines without meaning a special symbol. In this case you can change it using the routine:

```
fl_set_browser_specialkey(FL_OBJECT *ob, char key)
```

Remarks

See demo11.c for an example program using a `FL_NORMAL_BROWSER` to view files. demo28.c shows all different browsers. demo29.c shows the insertion and deletion of lines in a `FL_HOLD_BROWSER`.

Chapter 18

Other objects

18.1 Timer

Short description

Timer objects can be used to make a timer that runs down toward 0.0 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 .2 seconds.

Adding an object

To add a timer to a form you use the routine

```
FL_OBJECT *fl_add_timer(int type, float x, float y,
                        float w, float h, 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. Blinks if time expires.
FL_HIDDEN_TIMER	Not visible.

Interaction

When a visible timer expires it start blinking. The user can stop the blinking by pressing the mouse on it. The timer object is returned to the application program at the moment the time expires.

Other routines

To set the timer to a particular value use

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

delay gives the number of seconds the timer should run. Use 0.0 to reset the timer. To obtain the current delay left in the timer use

```
float fl_get_timer(FL_OBJECT *obj)
```

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

See demo34.c for the use of timers.

Part IV

Designing your own object classes

Chapter 19

Introduction

In a number of cases the functionality of the **Forms Library** might not be enough for your needs. Although free objects in principle provide you with all the flexibility you want, there are situation in which you would like to extend the library with new types of objects. For example, you might want to define switches or joysticks or other types of sliders, etc. Also, if you are not completely satisfied with the behaviour of some of the existing classes you might want to change them. Don't do this though by changing the original class. Make a copy instead and turn this into a new extra object class. In this way your library will remain compatible with future changes.

Creating such new object classes and adding them to the library is very simple. In fact it is almost the same as making free objects. This part gives you all the details of how to add new classes. In chapter 20 a global overview is given of how the **Forms Library** works and how it communicates with the different object classes by means of events. Chapter 21 describes in detail what type of events objects can receive and how they should react to them. Chapter 22 describes in detail the structure of the type `FL_OBJECT` which plays a crucial role. One of the important aspects of an object is how to draw it on the screen. Chapter 23 gives all the details on drawing objects. The **Forms Library** contains a large number of routines that help you drawing objects. In this chapter an overview is given of all of them. Finally, chapter 24 gives an example of a very simple object class.

The object classes you create might be useful to other people as well. Because the **Forms Library** is Public Domain software it is only fair that you make your own defined object classes Public Domain as well. So please send your new classes to us, using the Email address `markov@cs.ruu.nl`. We will see whether the class satisfies some standards and either include it in the next release or make it available through our archive.

Chapter 20

Global structure

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

The main module communicates with the object class modules by means of events. Each object has to have a handle routine associated which is called by the main module whenever something needs to be done. One of the arguments passed to the handle routine is the type of event, e.g. `FL_DRAW`, indicating that the object needs to be redrawn.

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

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

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

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

```
static int handle_NEW(FL_OBJECT *obj,int event,float mx,float my,char key)
```

This routine is the same as the handle routine for free objects and should handle particular events for the object. *mx*, *my* give the current mouse position and *key* the key that was pressed (if this information is related to the event). See chapter 21 for the types of events and the actions that should be taken. The routine should return whether the setting of the object is changed, i.e., whether the object has to be returned to the application program by `fl_do_forms()` or `fl_check_forms()`.

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

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

that sets particular values for the object and

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

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

20.1 The routine `fl_add_NEW()`

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

```
FL_OBJECT *fl_create_NEW(int type,float x,float y,
         float w,float h,char label[])
{
   FL_OBJECT *ob;
   ob = fl_make_object(FL_NEW,type,x,y,w,h,label,handle_NEW);
   /* fill in defaults */
   /* allocate special storage and fill it with defaults */
   return ob;
}

FL_OBJECT *add_NEW(int type,float x,float y,float w,float h,char label[])
{
   FL_OBJECT *ob;
   ob = fl_create_NEW(type,x,y,w,h,label);
   fl_add_object(fl_current_form,ob);
   return ob;
}
```

The constant `FL_NEW` will indicate the object class. It should be an integer. The numbers 0–1000 and 5000 and higher are reserved for the system and should not be used. The routine `fl_make_object()` creates the object and fills in its class, etc. The routine `fl_add_object()` has to be called at the end of the routine to actually add the object to the current form.

Chapter 21

Events

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

FL_DRAW The object has to be redrawn. To figure out the size of the object you can use the fields $obj \to x$, $obj \to y$, $obj \to w$ and $obj \to h$. Some other aspects might also influence the way the object has to be drawn. E.g., you might want to draw the object differently when the mouse is on top of it or when the mouse is pressed on it. The is can be figured out as follows. The field $obj \to belowmouse$ indicates whether the object is below the mouse. The field $obj \to pushed$ indicates whether the object is currently being pushed with the mouse. Finally, $obj \to focus$ indicate whether input focus is directed towards this object. Note that the drawing of the object is the full responsibility of the object class. So e.g. also the bounding box and the label, which can be found in the field $obj \to label$ have to be drawn by the object class. The **Forms Library** provides a large number of routines to help you draw object. See chapter 23 for more details on drawing objects and an overview of all available routines.

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

FL_LEAVE The mouse has left the bounding box. Again, normally a redraw is enough (or nothing needs to be done).

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

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

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

FL_MOUSE The mouse position has changed. This event is sent to an object between a `FL_PUSH` and a `FL_RELEASE` event. The mouse position is given with the routine and action can be taken. For example, sliders use this event while e.g. buttons do not.

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

FL_UNFOCUS Input is no longer focussed on this object.

FL_KEYBOARD A key was pressed. The ascii value is given with the routine. This event only happens between `FL_FOCUS` and `FL_UNFOCUS` events. 1, 2, 3 and 4 indicate the left, right, up and down arrow keys.

FL_STEP A step event is sent all the time (at most 100 times a second but often less because of e.g. time consuming redrawing) to an object if the field $obj \rightarrow automatic$ has been set to 1. This can be used to make object change appearance without user action. E.g. the clock uses these type of events.

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

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

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

21.1 Shortcuts

The **Forms Library** has a mechanism of dealing with keyboard shortcuts. In this way the user can use the keyboard rather than the mouse for particular actions. At the moment there are two object classes that use this, namely buttons and menus. The mechanism works as follows. There is a routine

```
void fl_set_object_shortcut(FL_OBJECT *obj, char str[])
```

with which the object class can bind a series of keys to an object. E.g., when *str* is `"acE#D^h"` the keys a,c,E, <Alt> D and <Contrl> h are associated to the object. The precise format here is as follows: Any character in the string is considered as a shortcut. Only `^` and `#` stand for combinations with the <Ctrl> and <Alt> key. (There is no difference between e.g. `^C` and `^c`.) The symbol `^` itself can be obtained using `^^`. The symbol `#` can be obtained using `^#`. So, e.g. `#^#` means <Alt>`#`.

Now whenever the user presses one of these keys an `FL_SHORTCUT` event is send to the object. Here the key pressed is given with the handle routine (in the argument *key*). Combinations with the <Alt> key are given by adding 128 to the ASCII value of the rest. E.g. `#^E` is passed as 132 (5+128). The object can now take action accordingly. For example, the menu type class allows the user to associate key strokes to menu items. (Of course it has to keep its own internal administration of which keys are associated to which items.) The idea is NOT that the user itself calls `fl_set_object_shortcut()` but that the class provides a routine for this that also keeps track of the required internal bookkeeping.

The order of handling keys is as follows: First it is tested whether any object in the form has the key as a shortcut. In this case the first of those objects gets the events. Otherwise, if the key is <Tab> or <Return> input is focussed on the next input field unless the current input object has its field $obj \rightarrow wantall$ set to 1. Otherwise, the key is sent to the current input field. Hence, note that input objects only get a <Tab> or <Return> key send to them if they set the field $obj \rightarrow wantall$. This is e.g. used in multiline input fields.

Chapter 22

The type FL_OBJECT

Each object contains a number of fields. Some of them are used by the main routine, some have a fixed meaning and should never be altered by the class routines and some are free for the class routines to use. Below we consider all of them.

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

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

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

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

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

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

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

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

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

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

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

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

visible Indicates whether the object is visible. The class routines don't have to do anything with this variable. When the object is not visible the main routine will never try to draw it or send events to it. By default objects are visible.

active Indicates whether the object is active, i.e., wants to receive events other than `FL_DRAW`. Changing the status should be done in the `fl_add_NEW()` routine if required. By default objects are active. Boxes and texts are not active.

input Whether this object can receive keyboard input. If not, events that are related to keyboard input are not sent to the object. Default input is false. It should be set by `fl_add_NEW()` if required.

wantall An input object normally does not receive <Tab> or <Return> keystrokes because these are used to switch between input objects. By setting this field to 1 these keystrokes will also be sent to the object when focus is directed to it.

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

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

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

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

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

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

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

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

next,prev,form These are pointers to other objects in the form and to the form itself. They are used by the main routines. The class routines should never change or use them.

Chapter 23

Drawing objects

An important aspect of a new object class (or a free object) is how to draw it. As indicated above this should happen when the event `FL_DRAW` is sent to the object. The place, i.e. bounding box, where the object has to be drawn is indicated by the fields $obj \rightarrow x$, $obj \rightarrow y$, $obj \rightarrow w$ and $obj \rightarrow h$. Forms are drawn either in RGB mode or in colormap mode, depending on the type of machine. Hence, preferably your classes should run well in both modes. If you have to make a choice though, choose RGB mode cause that is what will be used most. To find out whether the mode is RGB or not check the variable `fl_rgbmode`. If it is 1 you have to use RGBmode. To avoid having to write two drawing routines for two different modes a routine

```
void fl_color(short ind)
```

has been provided. When in colormap mode it sets the current color to index *ind*. When in RGBmode it sets the RGBcolor to the corresponding colormap color. To this end, the library keeps an internal colormap that changes colormap entries into RGB colors. This internal colormap is initialized with the system colormap. If you use this routine instead of the GL routine `color()` your drawing will work correct in both modes. (All drawing routines described below use this.) To set a particular color in the internal colormap use `fl_mapcolor()` and to read the values of a particular color index in the colormap use `fl_getmcolor()`. These routines replace the GL routines `mapcolor()` and `getmcolor()`. Most object classes currently available use these routines. Note that the internal colormap does not fully behave the same way as the system colormap. E.g. changing a value does not mean that the object automatically gets a new color. It has to be redrawn to achieve that.

The coordinate system of the form corresponds directly to the screen. Hence a pixel on the screen always has size 1 in the coordinate system of the form. Whenever the user changes the size of the form window on the screen all coordinates in the form are adapted to keep this property. Hence, when drawing you can always assume that the coordinate systems are the same. The obtain the position of the mouse in the current form, use the routine (The lower left corner of the form has coordinates (0,0).)

```
void fl_get_mouse(float *xx, float *yy)
```

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

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

It sets a clipping region. x, y, w and h are as in the definition of objects. Drawing is restricted to this region after the call. In this way you can avoid information running out of an object. Always use after drawing

```
void fl_unset_clipping()
```

to stop clipping.

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

```
float fl_get_char_height(float size, int style)
```

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

Returns the character height and width of the font used, where *size* indicates the pointsize for the font and *style* is the style in which the text is drawn.

```
float fl_get_string_width(float size, int style,char str[])
```

Returns the width of the string *str* in the font.

There are a number of routines that help you drawing objects on the screen. First of all there are a few low level routines:

```
void fl_rect(float x, float y, float w, float h, int col)
```

Draws a rectangle on the screen in color *col*. It has no border. To draw a rectangle with a black border use

```
void fl_rectbound(float x, float y, float w, float h, int col)
```

To only draw a border in some color use

```
void fl_bound(float x, float y, float w, float h, int col)
```

Finally there is a routine to draw a line. It draws the diagonal of the box given argument. So to draw a horizontal line make $h = 1.0$ not 0.0.

```
void fl_line(float x, float y, float w, float h, int col)
```

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

```
void fl_drw_box(int style, float x, float y, float w, float h, int c, int bw)
```

Draws a box. *style* is the type of the box, e.g `FL_DOWN_BOX`. x, y, w, and h, indicate the size of the box. c is the color. bw is the width of the boundary.

For drawing text there are three routines:

```
void fl_drw_text_cursor(int align, float x, float y, float w, float h,
         int c, float size, int style, char str[], int cc, int pos)

void fl_drw_text(int align, float x, float y, float w, float h,
         int c, float size, int style, char str[])

void fl_drw_text_beside(int align, float x, float y, float w, float h,
         int c, float size, int style, char str[])
```

align is the alignment, x, y, w and h indicate the bounding box, *c* is the color of the text, *size* is its size (in points), *style* is the style to be used, *str* is the string itself, *cc* is the color of the cursor and *pos* is the position of the cursor (-1 means show no cursor). `fl_drw_text()` is like `fl_drw_text_cursor` but without the cursor. `fl_drw_text_beside()` aligns outside the box rather than inside (as used for labels in almost all objects except text objects). The last two routines interpret a text string starting with the character `@` differently and draw some symbol instead. This is perfect for labels but it is possibly not what you want in other cases. If you don't want this behavior, always use `fl_drw_text_cursor()` (possibly without a cursor).

Chapter 24

An example

Let us work through an example of how to create a simple object class Colorbox. Assume we want a class with the following behavior: It should normally be red. When the user presses the mouse on it it should turn blue. When the user releases the mouse button again the object should turn red again and be returned to the application program.

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

```
#define FL_COLBOX 1001 /* Class number (>1000, <5000) */

#define FL_NORMAL_COLBOX 0 /* The only type */
```

Normally it should also contain some defaults for e.g. the boxtype and label alignment). The include file also has to define the routines available for this object class. I.e., it should contain:

```
FL_OBJECT *fl_create_colbox(int, float, float, float, float, char []);
FL_OBJECT *fl_add_colbox(int, float, float, float, float, char []);
```

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

```
FL_OBJECT *fl_create_colbox(int type, float x, float y, float w, float h,
                                char label[])
{
  FL_OBJECT *ob;
  ob = fl_make_object(FL_COLBOX,type,x,y,w,h,label,handle_colbox);
  ob->col1 = 1; /* Red */
  ob->col2 = 4; /* Blue */
  return ob;
}

FL_OBJECT *fl_add_colbox(int type, float x, float y, float w, float h,
                                char label[])
{
```

```
  FL_OBJECT *ob = fl_create_colbox(type,x,y,w,h,label);
  fl_add_object(fl_current_form,ob);
  return ob;
}
```

The fields *col*1 and *col*2 are used to store the two colors red and blue such that the user can change them when required with the routine `fl_set_object_color()`. What remains is to write the handling routine `handle_colbox()`. It has to react to three types of events: `FL_DRAW`, `FL_PUSH` and `FL_RELEASE`. Note that the fact whether the mouse is pushed on the object is indicated in the field $ob \rightarrow pushed$. Hence, when pushing and releasing the mouse the only thing that needs to be done is redrawing the object. This leads to the following piece of code:

```
static int handle_colbox(FL_OBJECT *ob, int event,
                                float mx, float my, char key)
{
  switch (event) {
   case FL_DRAW:
    /* Draw box */
    if (ob->pushed)
     fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col2,FL_BOUND_WIDTH);
    else
     fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col1,FL_BOUND_WIDTH);
    /* Draw label */
    fl_drw_text_beside(ob->align,ob->x,ob->y,ob->w,ob->h,
                              ob->lcol,ob->lsize,ob->lstyle,ob->label);
    return 0;
   case FL_PUSH:
    fl_redraw_object(ob);
    return 0;
   case FL_RELEASE:
    fl_redraw_object(ob);
    return 1;
  }
  return 0;
}
```

That is the whole piece of code.

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

Part V

Appendices

Appendix A

Overview of main routines

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

A.1 Initialization

The routine

```
void fl_init()
```

initializes the **Forms Library**. It normally needs not be called unless you use routines from the library before defining a form or using a goody.

Based on the machine configuration, the library determines whether to run in colormap or RGB mode and single or double buffered mode. This can be set by using the routine

```
void fl_set_graphics_mode(int rgbmode,int doublebuf)
```

rgbmode indicates whether RGB mode should be used, *doublebuf* indicates whether double buffering should be used. Call this routine before anything else and don't change the modes during the program. Changing from double to single buffered mode might in particular be useful. Many forms still look good when in single buffered mode. When you forms control some continuous process single buffering will speed up the process considerable because there is no need to wait for vertical blanks (which might take 1/60 th of a second) each time something changes in the form.

Normally the **Forms Library** reports errors. This can be avoided using the routine

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

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

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

```
void fl_set_font_name(int numb,char name[])
```

See section 3.10.3 for details. A redraw of all forms is required to actually see the change.

A.2 Creating forms

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

Starts the definition of a form. *type* is the type of the box that is used as a background. *w* and *h* give the width and height of the form. Returns a pointer to the form.

```
void fl_end_form()
```

End the definition of a form.

```
FL_OBJECT *fl_bgn_group()
```

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

```
void fl_end_group()
```

Ends the definition of a group.

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

Reopens a form for adding objects to it.

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

Removes an object from the form it is in.

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

Frees the memory for an object. (Object should be deleted first.)

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

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

A.3 Doing interaction

```
long fl_show_form(FL_FORM *form,int place,int border,char name[])
```

Displays a form on the screen. *place* controls the position and size of the form. *border* indicates whether a border should be drawn around the form. In this case *name* is the name of the window. The routine returns the window identifier of the form. Multiple forms can be displayed at the same moment. When placing a form on the screen using place `FL_PLACE_POSITION` the position and size can be set using the routines

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

```
void fl_set_form_size(FL_FORM *form,int w,int h)
```

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

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

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

Hides the particular form, i.e., closes its window.

```
FL_OBJECT *fl_do_forms()
```

Does the interaction with the currently displayed forms. The routine ends when the state of some object changes. A pointer to this object is returned.

```
FL_OBJECT *fl_check_forms()
```

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

```
FL_OBJECT *fl_do_only_forms()
```

```
FL_OBJECT *fl_check_only_forms()
```

Same but these routines do not handle user events.

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

Activates a form for user interaction.

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

Deactivates a form to stop user interaction with it.

```
void fl_deactivate_all_forms()
```

```
void fl_activate_all_forms()
```

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

```
void fl_activate_object(FL_OBJECT *obj)
```

Activates an object for user interaction.

```
void fl_deactivate_object(FL_OBJECT *obj)
```

Deactivates an object to stop user interaction with it.

```
void fl_redraw_object(FL_OBJECT *obj)
```

This routine redraws the particular object. If *obj* is a group it redraws the complete group. Normally you should never need this routine because all library routines take care of redrawing but there might be situations in which a redraw is required.

```
void fl_redraw_form(FL_FORM *form)
```

Redraws an entire form.

A.4 Setting attributes

A number of general routines are available for setting attributes:

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

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

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

Changes the shape of the bounding box of the object.

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

void fl_set_object_lsize(FL_OBJECT *obj,float lsize)

void fl_set_object_lstyle(FL_OBJECT *obj,int lstyle)

void fl_set_object_align(FL_OBJECT *obj,int align)

void fl_set_object_label(FL_OBJECT *obj,char label[])
```

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

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

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

and

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

The routine

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

binds a call-back routine to an object.

```
void fl_set_form_call_back(FL_FORM *form,void (*callback)())
```

Binds a call-back routine to an entire form.

```
void fl_set_event_call_back(void (*callback)())
```

Sets up a call-back routine for all user events.

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

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

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

Makes the object or group invisible.

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

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

Appendix B

Trouble Shooting

This appendix deals with a number of (common) problems encountered by people using the **Forms Library**. Ways of avoiding them are presented.

The program is very slow On machines with more than 8 bits for colt the **Forms Library** runs in double buffered mode. This means that whenever something changes in a form the program waits for a vertical blank which happens only 60 times a second. In particular when the program e.g. fills in a number of fields in the form, after each fillin the program will wait 1/60 second. This can be avoided in a number of ways. First of all, it is possible to use the routines `fl_freeze_form()` and `fl_unfreeze_form()` to temporarily stop the redisplaying of this form. Changing the values between the freezing calls result in only once waiting for a vertical blank.

A second solution is to run the package in single buffer mode. To this end use the call `fl_set_graphics_mode(1,0)` before displaying any forms (the first argument indicates whether to use RGB mode, the second whether to work in double buffered mode). It is also possible to change the double buffering to single buffering for a single form by setting `form->doublebuf = 0`. Of course forms look slightly less nice in single buffered mode.

The program does not see the events you queued There can be two reasons for this. One cause might be that you did not use the replacement routines like `fl_qdevice()` instead of the GL routines like `qdevice()`. It is also possible that you queued the devices before calling the initialization routine `fl_init()` or you forgot calling this routine at all.

The program does not immediately react to user events In some cases the program suddenly stops reacting correctly to user events or runs behind (e.g. all the time reacts to the previous event, not the current). This is caused by a mistake in the application program in handling events. First of all, check that none of the GL event handling calls (like `qtest()`) are used but always their replacements (like `fl_qtest()`). Secondly take care that whenever the object `FL_EVENT` is reported an event is read from the event queue (using `fl_qread()`). For each event placed in the event queue only once this object is returned. A typical way in which this goes wrong is the following: The main program places e.g. a form on the screen and its own window. Next it calls `fl_do_forms()` and, takes action when objects are returned (reading events when `FL_EVENT` is returned). Now at some stage it calls a routine that temporarily displays another form (e.g. to get some value from the user) and waits (again using `fl_do_forms()`) until the user presses the OK button on this form. This seems correct but it is not. This routine might get an `FL_EVENT` object returned but does not react to it (it does not even take it out of the queue). Now the application program will run one event behind the queue, causing the problem. The routine should either handle (or throw away) these events or use `fl_do_only_forms()` instead which never returns events.

The safest way though of dealing with user events is using an event callback routine with `fl_set_event_callback()`. Be carefull though. When you use an event callback routine NEVER call `fl_qtest()` or `fl_qread()` especially not from within the event callback routine. It will lead to an infinite loop.

The boxes of the objects have a strange left-bottom border This happens only under some versions of the operating system on some particular machines. It has to do with roundoff errors in the placement of pixels on the screen. It especially occurs when forms are scaled. So avoid scaling forms as much as possible.

Reporting Bugs

When you (think you) encountered a bug in the **Forms Library** please report it by sending a mail message to `markov@cs.ruu.nl`. In this mail please indicate the type of machine you are running this on, the type of graphics option, the version of IRIX and the version of **Forms Library**. E.g.: Personal Iris 4D-35 TG running IRIX 4.0.5F and Forms 2.2a. You can find the version of the **Forms Library** by holding the <Pause> and pressing the left mouse button somewhere in one of the forms. Give a short description of the problem and if possible, some sample code that exhibits the behaviour. Don't expect an immediate answer but I will do my best.