Forms Library

A Graphical User Interface Toolkit for X

Forms Library
A Graphical User Interface Toolkit for X
V0.81 July 1996

T. C. Zhao†
Department of Physics, University of Wisconsin-Milwaukee
Milwaukee, WI 53201, USA

and

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

Preface

Window-based user interfaces are becoming a common and required feature for most computer systems, and as a result, users have come to expect all applications to have polished user-friendly interfaces. Unfortunately, constructing user interfaces for programs is in general a time consuming process. In the last few years a 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 capabilities. The **Forms Library** was constructed to remedy this problem. The design goals when making the **Forms Library** were to create a package that is intuitive, 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 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 exist, like buttons (of many different flavors) that the user can push with the mouse, sliders with which the user can indicate a particular setting, input fields in which the user can provide textual input, menus 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 callback 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 be fine tuned to one's 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 only the services provided by the *Xlib* and should run on all workstations that have X installed on them. The current version needs 4bits of color (or grayscale) to look nice, but it will function properly on workstations having less depth (e.g., **XForms** works on B&W X-terminals).

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 show how easy forms are built and used. For simple forms and those that may be frequently used in application programs, e.g., to ask a question or select a file name, special routines are provided. For example, to let the user choose a file in a graphical way (allowing him/her to walk 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 generate 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.

Although this document describes all you need to know about using the **Forms Library for X**, it is not an X tutorial. On the contrary, details of programming in X are purposely hidden in the **Forms Library** interfaces, and one need not be an X-expert to use the **Forms Library**, although some knowledge of how X works would help to understand the inner workings of the **Forms Library**.

Forms Library and all the programs either described in this document or distributed as demos have been tested under X11 R4, R5 & R6 on all major UNIX platforms, including SGI, SUN, HP, IBM RS6000/AIX, Dec Alpha/OSF1, as well as PC/Linux, PC/FreeBSD/NetBSD, SCO and Unixware.

This document consists of four parts. The first part is a tutorial that provides an easy, informal introduction to the **Forms Library**. This part should be read by everybody that wants to use the library. You are encouraged to try variations of the demo programs distributed in the **Forms Library** package.

Part II describes the **Form Designer** with which you can design forms interactively and have **Form Designer** write code for you.

Part III 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 IV describes how this should be done.

Version Note

The authors request that the following name(s) be used when referring to this toolkit

XForms
Forms Library for X
or simply
Forms Library

Figures in this document were produced by `fd2eps`, a program that takes the output of the form designer and converts the form definition into an encapsulated POSTSCRIPT file.

This document is dated June 29, 1996.

Support

Although **XForms** has gone through extensive testing, there are most likely a number of bugs remaining. Your comments would be greatly appreciated. Please send any bug reports or suggestions to T.C. Zhao (`zhao@bragg.phys.uwm.edu`). Please do not expect an immediate response, but we do appreciate your input and will do our best.

Bindings to other languages

As of this writing, `perl`, `python`, and `Ada95` bindings to the **Forms Library** are in beta. Follow the links on **XForms**'s home page to get more info on these bindings.

Archive Sites

Permanent home for the **Forms Library** is at

```
ftp://laue.phys.uwm.edu/pub/xforms
ftp://ftp.cs.ruu.nl/pub/XFORMS  (mirror site)
ftp://imageek.york.cuny.edu/xforms  (mirror site)
```

For additional mirrors, html version of this document, news and other info related to **XForms** can be accessed through www via the following URL

```
http://bragg.phys.uwm.edu/xforms
```

In addition to ftp and www server, a mail server is available for those who do not have direct internet access.

To use the mail server, send a message to `mail-server@cs.ruu.nl` or the old-fashioned path alternative `uunet!mcsun!sun4nl!ruuinf!mail-server`.

The message should be something like the following

```
begin
path fred@stone.age.edu  (substitute your address)
send help
end
```

To get a complete listing of the archive tree, issue `send ls-lR.Z`.

Mailing List

A mailing list for news and discussions about **XForms** is available. To subscribe or un-subscribe, send a message to `listserv@ra.york.cuny.edu` with one of the following commands as the mail body

```
help
subscribe   xforms
unsubscribe xforms
```

To use the mailing list, send mail to `xforms@ra.york.cuny.edu`. Please remember that the message will be sent to hundreds of people. Please **Do not** send subscribe/unsubscribe messages to the mailing list, send them to `listserv@ra.york.cuny.edu`.

† Currently GE Medical Systems, P.O. Box 414, NB-922, Milwaukee, WI 53201-0414

List of Figures

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. Following the X tradition, **Forms Library** does not enforce the look and feel of objects although in its default state, it does provide a consistent look and feel for all objects.

The **Forms Library** only uses the services provided by Xlib and should be compilable on all machines that have X installed and have an ANSI compatible compiler. Being based on Xlib, **Forms Library** is small and efficient. It can be used in both C and C++ programs and soon it will be available for other languages.[1]

The basic procedure 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 that the object changed) and the program can take action accordingly, after which control is returned again to the `fl_do_forms()` routine. Multiple forms can be handled simultaneously by the library and can be combined with windows of the application program. More advanced event handling via object callbacks is also supported.

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 show how to piece together various parts of the library and demonstrate how easy forms are built and used. They can be found in the directory `xforms/DEMOS`. Studying these demos is a good way of 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. As its name implies, this is a

[1] As of this writing, `perl`, `Ada95` and `python` bindings are in beta testing

program that lets you interactively design forms and generate the corresponding C-code. See Part II 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 describe the basic application programmer's interface to the **Forms Library** and 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 III of this document. Chapter 4 describes how to set up interaction with forms. A very specific class of objects are free objects and canvases. 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 built-in routines for simple interaction like asking questions and prompting for choices etc.

Chapter 2

Getting Started

2.1 Naming Conventions

The names of all **Forms Library** functions and user-accessible data structures begin with `fl_` or `FL_`, and use an "underscore-between-words" convention, that is when function and variable names are composed of more than one word, an underscore is inserted between each word. For example,

```
fl_state
fl_set_object_label()
fl_show_form()
```

All **Forms Library** macros, constants and types also follow this convention, except that the first two letters are capitalized. For example,

```
FL_min()
FL_NORMAL_BUTTON
FL_OBJECT
```

2.2 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,230,160);
   fl_add_button(FL_NORMAL_BUTTON,40,50,150,60,"Push Me");
fl_end_form();
```

The first line indicates the start of the form definition. `simpleform` will later be used to identify the form. The type of the form is `FL_UP_BOX`. This means that the background of the form is a raised box that looks like it is coming out of the screen (See Fig. 2.1). The form has a size of 230 by 160 pixels[1]. 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 the virtue of the button geometry supplied 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,FL_NOBORDER,"SimpleForm");
```

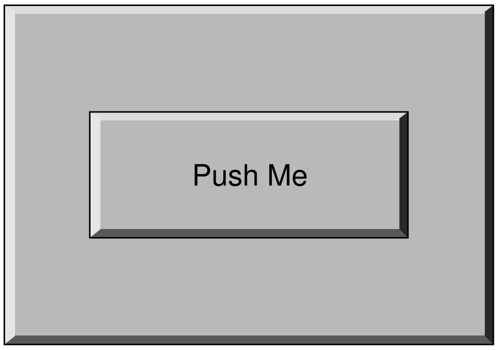

Figure 2.1: Simple button

This will show the form on the screen at the mouse position. (The third argument indicates whether the form gets window manager's decoration and the fourth is the window title.)

Next we give the control over the interaction to the **Forms Library**'s main event loop using the call

```
fl_do_forms(void);
```

This will handle interaction with the form until you press and release the button with the mouse at which moment control is returned to the program. Now the form can be removed from the screen (and have its associated window destroyed) using

```
fl_hide_form(simpleform);
```

The complete program is given in the file `demo01.c` in the directory `xforms/DEMOS`. All demonstration programs can be found in this directory. Studying them is a good way of learning how the library works.

Compile and run it to see the effect. To compile a program using the **Forms Library** use the following command or something similar

[1] **Forms Library** also supports screen resolution independent size specifications where sizes are given in milli-meter, point (1/72 inch) or 100th of a mm or point

```
cc -I../FORMS -O -o demo01 demo01.c -L../FORMS -lforms -lX11 -lm
```

(Of course you can install the library so that `-L../FORMS` and `-I../FORMS` can be omitted. Contact your systems administrator or read the `Readme` file in the directory `xforms` 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 program can be found in `yesno.c`.)

```
#include "forms.h"

FL_FORM *form;
FL_OBJECT *yes, *no, *but;
main(int argc, char *argv[])
{

   fl_initialize(&argc, argv, "FormDemo", 0, 0);
   form = fl_bgn_form(FL_UP_BOX, 320, 120);
     fl_add_box(FL_NO_BOX, 160, 40, 0, 0, "Do you want to Quit?");
     yes = fl_add_button(FL_NORMAL_BUTTON, 40, 70, 80, 30,"Yes");
     no  = fl_add_button(FL_NORMAL_BUTTON, 200, 70, 80, 30,"No");
   fl_end_form();
   fl_show_form(form,FL_PLACE_MOUSE,FL_TRANSIENT,"Question");
   while((but = fl_do_forms()) != yes)
      ;
   fl_hide_form(form);
   return 0;
}
```

It creates a form with a simple text and two buttons (See Fig 2.2). 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.

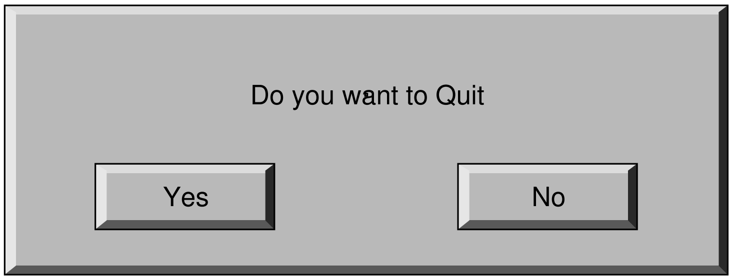

Figure 2.2: A simple question

As you see, the program starts by calling the routine `fl_initialize()`. This routine should be called before any other calls to the library are made (except for `fl_set_defaults()`). One of the things this routine does is to establish a connection to the X server and initialize a resource database used by the X resource manager. It also does many other things, such as parsing command line options and initializing internal **Forms Library** structures. For now, it suffices to know that by calling this routine, a program automatically recognizes the following command line options

Options	Value type	Meaning
`-display` *host:dpy*	string	Remote host
`-name` *appname*	string	change application name
`-visual` *class*	string	TrueColor, PseudoColor etc
`-depth` *depth*	int	Prefered visual depth.
`-private`	none	Force a private colormap
`-shared`	none	Always share colormap
`-stdcmap`	none	Use standard colormap
`-debug` *level*	integer	Print some debug information
`-sync`	none	Force synchronous mode

See Appendix A 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.

The above program only shows one of the event handling methods provided by the library. The direct method of event handling shown above is appropriate for simple programs. For reasonably complicated ones, however, utilizing object callback is strongly encouraged.

We demonstrate the use of object callbacks using the previous example with some modifications so that event processing via callbacks is utilized. It is recommended and also typical of a good **XForms**application to **separate** the UI components and the application program itself. Typically the UI components are generated by the bundled GUI builder and the application program consists mostly of callbacks and some glue that combines the UI and the program.

To use callbacks, a typical procedure would be to define all the callback functions first, then register them with the system using `fl_set_object_callback()`. After the form is realized (shown), control is handed to **Forms Library**'s main loop `fl_do_forms()`, which responds to user events indefinitely and never returns.

After modifications are made to utilize object callbacks, the simple question example looks as follows:

```
#include "forms.h"

void yes_callback(FL_OBJECT *ob, long user_data)
{
    printf("Yes is pushed\n");
```

```
    fl_finish();
    exit(0);
}

void no_callback(FL_OBJECT *ob, long user_data)
{
    printf("No is pushed\n");
}


FL_FORM *form;
main(int argc, char *argv[])
{
   FL_OBJECT *obj;

   fl_initialize(&argc, argv, ,"FormDemo", 0, 0);
   form = fl_bgn_form(FL_UP_BOX, 320, 120);
      fl_add_box(FL_NO_BOX, 160, 40, 0, 0, "Do you want to Quit?");
      obj = fl_add_button(FL_NORMAL_BUTTON, 40, 70, 80, 30,"Yes");
      fl_set_object_callback(obj, yes_callback, 0);
      obj  = fl_add_button(FL_NORMAL_BUTTON, 200, 70, 80, 30,"No");
      fl_set_object_callback(obj, no_callback, 0);
   fl_end_form();
   fl_show_form(form,FL_PLACE_MOUSE,FL_TRANSIENT,"Question");
   fl_do_forms();
}
```

In this example, callback routines for both the `yes` and `no` buttons are first defined. Then they are registered with the system using `fl_set_object_callback()`. After the form is shown, the event handling is again handed to the main loop in **Forms Library** via `fl_do_forms()`. In this case, whenever the buttons are pushed, the callback routine is invoked with the object being pushed as the first argument to the callback function, and `fl_do_forms()` never returns.

You might also notice that in this example, both buttons are anonymous, that is, it is not possible to reference the buttons directly outside of the creation routine. This is often desirable when callback functions are bound to objects as the objects themselves will not be referenced except as callback arguments. By creating anonymous objects, a program avoids littering itself with useless identifiers.

2.3 Programming Model

To summarize, every **Forms Library** application program must perform several basic steps. These are

`Initialize the` **Forms Library** This step establishes a connection to the X server, allocates resources and otherwise initializes the **Forms Library**'s internal structures, which include visual selection, font initialization and command line parsing.

`Defining forms` Every program creates one or more forms and all the objects on them to construct the user interface. This step may also include callback registration and per object initialization such as setting bounds for sliders etc.

`Showing forms` This step makes the designed user interface visible by creating and mapping the window (and subwindows) used by the forms.

`Main loop` Most **Forms Library** applications are completely event-driven, and are designed to respond to user events indefinitely. The **Forms Library** main loop, `fl_do_forms()`, retrieves events from the X event queue, dispatches the retrieved event through appropriate objects, and notifies the application of what action, if any, should be taken. The actual notification methods depend on how the interaction is set up, which could be object callback or by returning to the application program the object whose status has changed.

The following chapters will lead you through each step of the process with more details.

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 III 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 II). Still it is useful to read through this chapter because it explains 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, FL_Coord w, FL_Coord h)
```

`w` and `h` indicate the width and height of the form (in pixels by default), i.e., the largest x- and y-coordinate that can be used in the form. Positions in the form will be indicated by integers between `0` and `w-1` or `h-1`. 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 meanings. The routine returns a pointer to the form just defined. This pointer must be used, for example, when drawing the form or doing interaction with it. The form definition ends with

```
void fl_end_form(void)
```

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,FL_Coord x,FL_Coord y,
                      FL_Coord w,FL_Coord h, const 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_OVAL_BOX`	A box shaped like an ellipse .
`FL_NO_BOX`	No box at all, only a centered label.

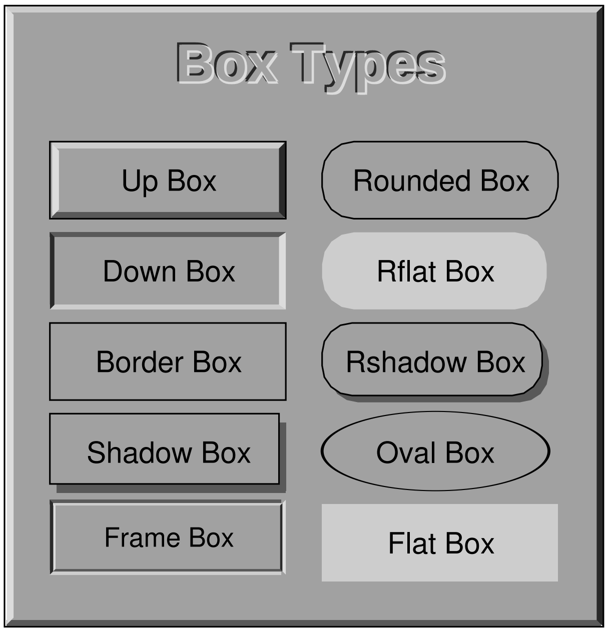

Figure 3.1: All boxtypes

`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 obtain multi-line labels, insert newline characters (`\n`) in the label string. It is also possible to underline the label or one of the characters in the label. This is accomplished by by embedding <CNTRL> `H` (`\010`) after the letter that needs to be underlined. If the first character of the label is <CNTRL> `H`, the entire label is underlined:

`u\010nderl\010ined` $\longrightarrow$ underlinded
`\010underlined` $\longrightarrow$ underlined

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 later references to the object.

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 3.11. 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,20,20,100,100,"I am a Box");
fl_set_object_color(thebox,FL_RED,0);       /* make box red */
fl_set_object_lcol(thebox,FL_GREEN);        /* make label green */
fl_set_object_lsize(thebox,FL_LARGE_SIZE);  /* 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,FL_Coord x,FL_Coord y,
                       FL_Coord w,FL_Coord h,const 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 in section 3.11. Again, the text can be multi-lined or underlined by embedding respectively the newline (`\n`) or <CNTRL> `H` (`\010`) in the label. The style, size and color of the text can be controlled and changed in many ways. See section 3.11.

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 put different lines of text below each other. Labels inside boxes are by default centered in the box. You can change the position of the text inside the box using the routines in section 3.11. 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.

3.4 Buttons

A very important class of objects are 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,FL_Coord x,FL_Coord y,
                         FL_Coord w,FL_Coord h, const 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 ones are:

```
FL_NORMAL_BUTTON
FL_PUSH_BUTTON
FL_TOUCH_BUTTON
FL_RADIO_BUTTON
```

They all look the same on the screen but their functions are 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 returns to its normal position 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 a choice among some mutually exclusive possibilities.

Whenever the user pushes a button and then releases the mouse, the interaction routine `fl_do_forms()` is interrupted and returns a pointer to the button that was pushed and released. If a callback routine is present for the object being pushed, this routine will be invoked. In either case, the application program knows that the button was pushed and can take action accordingly. In the first case, control will have to be returned to `fl_do_forms()` again after the appropriate action is performed; and in the latter, `fl_do_forms()` would never return. 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` indicates 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 `pushbutton.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 3.11.

There are 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. The color of the light can be controlled with the routine `fl_set_object_color()`. See section 3.11.

`Round buttons` are buttons that are round. Use `fl_add_roundbutton()` to add a round button to a form. Both round and light buttons are nice as radio buttons.

`Check buttons` are buttons that have a small checkbox the user can push. To add a check button, use `fl_add_checkbutton()`. More stylish for a group of radio buttons.

`Bitmap buttons` are buttons that have bitmaps on top of the box. Use routine `fl_add_bitmapbutton()` to add a bitmap button to a form.

`Pixmap buttons` are buttons that have pixmaps on top of the box. Use routine `fl_add_pixmapbutton()` to add a pixmap button to a form.

Playing with different boxtypes, colors, etc., you can make many different types of buttons. See `buttonall.c` for some examples. Fig. 14.1 shows all buttons in their default states.

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

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

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

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

Whenever the value of the slider is changed by the user, it results in the slider being returned to the application program or the callback routine invoked. The program can read the slider value using the call

```
double 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 ValSliders

`Valslider` is almost identical with a normal slider. The only difference is the way the slider is drawn. For valsliders, in addition to the slider itself, its current value is also shown.

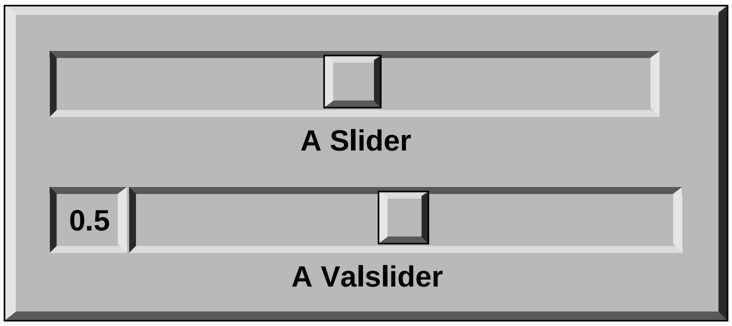

Figure 3.2: Slider and Valslider

3.7 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,FL_Coord x,FL_Coord y,
                        FL_Coord w,FL_Coord h,const char *label)
```

The main type of input field available is `FL_NORMAL_INPUT`. The input field normally looks like an `FL_DOWN_BOX`. This can be changed using the routine `fl_set_object_boxtype()` to be described in section 3.11.

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. Full *emacs*(1) style editing is supported. When the user presses <RETURN> or <TAB> the input field is returned to the application program and further input is directed to the next input field. (The <RETURN> key only works if there

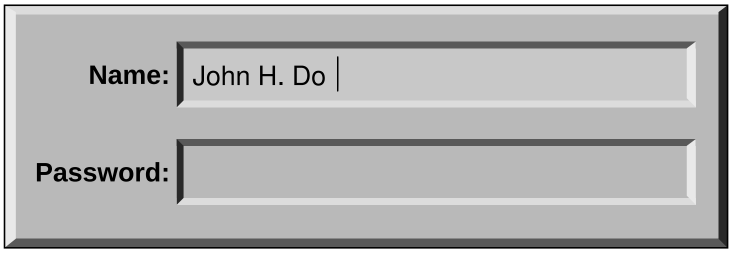

Figure 3.3: Input fields

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 erase character or replaced by any new input the user types in. Also the location of the cursor can be moved in the input field using the mouse.

The input field is fully integrated with the X Selection mechanism. Use the left button to cut from and the middle button to paste into an input field.

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

```
void fl_set_focus_object(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 (by 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,const 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:

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

Notice the bracket around the qualifier `const`. This indicates although the function is not declared to return a pointer to `const` string, it should be used as one. This is done mainly for compilation on machines whose string library header is buggy. Modifying the string returned by this function can produce unpredictable results.

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

3.8 Grouping objects

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

```
FL_OBJECT *fl_bgn_group(void)
```

and

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

Groups should never be nested. Groups are useful for two reasons. First of all it is possible to hide groups of objects (see section 3.9 below). This is often very handy. We can, for example, display part of a form only when the user asks for it. Some attributes are naturally multi-objects, e.g., to glue several objects together using the gravity attribute. Instead of setting the gravity for each object, you can place all related objects inside a group and set the resize/gravity attribute of the group.

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

It is possible to add objects to an existing group

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

where `group` is the object returned by `fl_bgn_group()`. After this call, you can start adding objects to the group (e.g., `fl_add_button` etc). The newly added objects are appended at the end of the group. When through adding, use `fl_end_group` as before.

3.9 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 are 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 only one of them is visible.

3.10 Deactivating, reactivating and triggering objects

Sometimes you might want a particular object to be temporarily inactive, 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, for example, be done by changing the label color to grey (see below.)

It is possible to simulate the action of an object being triggered from within the program by using the following routine

```
void fl_trigger_object(FL_OBJECT *)
```

Calling this routine on an object results in the object returned to the application program or its callback called if it exists. Note however, there is no visual feedback, i.e., `fl_trigger_object(button)` will not make the button appear pushed.

3.11 Changing attributes

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

3.11.1 Color

To change the color of a particular object use the routine

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

`col1` and `col2` are indices into a colormap. Which colors are actually changed depend on the type of objects. For box 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 bitmap buttons, `col1` is the color of the box and `col2` is the color of the bitmap. 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`. **Form Designer** comes in very handy in familiarizing you with various attributes since you can change all attributes of an object and immediately see the difference by "test"ing the object.

The following pre-defined color symbols can be used in all color change requests. If the workstation does not support this many colors, substitutions with the closest color will be made.

Name	RGB triple
FL_BLACK	(0, 0, 0)
FL_RED	(255, 0, 0)
FL_GREEN	(0,255, 0)
FL_YELLOW	(255,255, 0)
FL_BLUE	(0, 0,255)
FL_CYAN	(0,255,255)
FL_MAGENTA	(255, 0,255)
FL_WHITE	(255,255,255)
FL_COL1	(161,161,161)
FL_MCOL	(191,191,191)
FL_TOP_BCOL	(204,204,204)
FL_BOTTOM_BCOL	(89, 89, 89)
FL_RIGHT_BCOL	(51, 51, 51)
FL_LEFT_BCOL	(222,222,222)
FL_INACTIVE_COL	(110,110,110)
FL_TOMATO	(255, 99, 71)
FL_INDIANRED	(198,113,113)
FL_SLATEBLUE	(113,113,198)
FL_DARKGOLD	(205,149, 10)
FL_PALEGREEN	(113,198,113)
FL_ORCHID	(205,105,201)
FL_DARKCYAN	(40,170,175)
FL_DARKTOMATO	(139, 54, 38)
FL_WHEAT	(255,231,155)
FL_FREE_COL1	(?, ?, ?)

In the above table, `FL_FREE_COL1` has the the largest numerical value, and all color indices smaller than that are used (or can potentially be used) by the **Forms Library** although if you wish, they can also be changed using the following routine prior to `fl_initialize`:

```
void fl_set_icm_color(FL_COLOR index, int r, int g, int b)
```

Note that although the color of an object is indicated by a single index, it is not necessarily true that the **Forms Library** is operating in `PseudoColor`. **Forms Library** is capable of operating in all visuals and as a matter of fact the **Forms Library** will always select `TrueColor` or `DirectColor` if the hardware is capable of it. The actual color is handled by an internal colormap of `FL_MAX_COLS` entries (default 1024). To change or query the values of this internal colormap use the call

```
void fl_set_icm_color(FL_COLOR index, int r, int g, int b)
void fl_get_icm_color(FL_COLOR index, int *r, int *g, int *b)
```

Call `fl_set_icm_color` before `fl_initialize` to change **XForms**'s default colormap. Note these two routines do not communicate with the X server, they only populate/return information about the internal colormap, which is made known to the X server by the initialization routine `fl_initialize`.

To change the colormap and make a color index active so that it can be used in various drawing routines, use the following function

```
unsigned long fl_mapcolor(FL_COLOR i, int red, int green, int blue);
```

This function frees the previous allocated pixel and re-allocate the pixel with the RGB value specified. The pixel value is returned by the function. It is recommended that you use index larger than `FL_FREE_COL1` for your remap request to avoid accidentally free the colors you have not explicitly allocated.

Sometimes it may be more convenient to associate an index with a colorname, e.g., `"red"` etc., which may have been obtained via resources. To this end, the following routine exists

```
long fl_mapcolorname(FL_COLOR i, const char *name)
```

where varname is the color name[1]. The function returns -1 if the colorname `name` is not resolved.

You can obtain the RGB values of an index by using the following routine

```
unsigned long fl_getmcolor(FL_COLOR i, int *red, int *green, int *blue);
```

Function returns the pixel value as known by the Xserver.

Note that the current version only uses the lower byte of the primary color. Thus all primary colors in the above functions should be specified in the range of 0–255 inclusive.

To free any colors that you no longer need, the following routine should be used

```
void fl_free_colors(FL_COLOR colors[], int ncolors)
```

Prior to version 0.76, there is a color "leakage" in the implementation of the internal colormap that prevents the old index from being freed in the call `fl_mapcolor(index, r, g, b)`, resulting in . . accelerated colormap overflow and some other undesired behavior. Since there are many applications based on older versions of the **Forms Library**, a routine is provided to force the library to be compatible with the (buggy) behavior:

```
void fl_set_color_leak(int flag);
```

[1] Standard color names are listed in a file named `rgb.txt` and usually resides in `/usr/lib/X11`

Due to the use of an internal colormap and the simplified user interface, changing the colormap value for the index `index` may not result in a change of the color for the object. An actual redraw of the object (see below) whose color is changed may be required to have the change take effect. Therefore, a typical sequence of changing the color of a visible object is as follows:

```
fl_mapcolor(newcol, red, green, blue)  /* obj uses newcol */
fl_redraw_object(obj);
```

3.11.2 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. If you must do so, redraw the entire form after changing the boxtype of an object (see below). See the program `boxtype.c` for the effect of the boxtype on the different classes of objects.

It is possible to alter the appearance of an object by changing the border width attribute

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

Border width controls the "height" of an object, e.g., a button having a border width of 3 pixels appears more pronounced than one having a border width of 2 (see Fig 3.4). The **Forms Library**'s default is `FL_BOUND_WIDTH`(3) pixels. Note that the border width can be negative. Negative border width does not make a down box, rather, it makes the object having an upbox appear less pronounced and "softer". See program `borderwidth.c` for the effect of border width on different objects.

3.11.3 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,FL_COLOR lcol)
```

It sets the color of the label. The default is black (`FL_BLACK`). The font size of the label can be changed using the routine

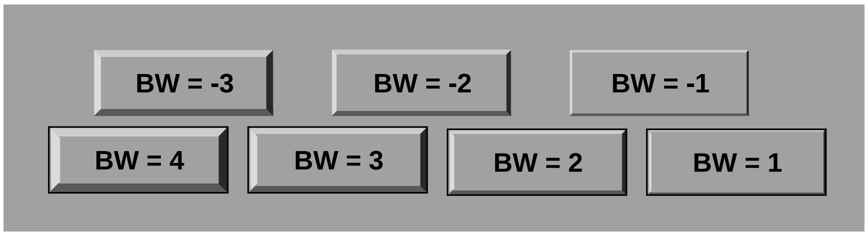

Figure 3.4: Object Border Width

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

`lsize` gives the size in points. Depending on the server and fonts installed, arbitrary sizes may or may not be possible. Fig 3.5 shows the font sizes that are standard with MIT/XConsortium distribution. So use of these values is encouraged. In any case, if a requested size can not be honored, substitution will be made. The default size for **XForms** is 10pt.

FL_TINY_SIZE	8pt	fl_tiny_size
FL_SMALL_SIZE	10pt	fl_small_size
FL_NORMAL_SIZE	12pt	fl_normal_size
FL_MEDIUM_SIZE	14pt	fl_medium_size
FL_LARGE_SIZE	18pt	fl_large_size
FL_HUGE_SIZE	24pt	fl_huge_size

Figure 3.5: Standard Font Sizes

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 default font for the **Forms Library** is Helvetica at 10pt. Additional styles are available:

`FL_NORMAL_STYLE`	Normal text
`FL_BOLD_STYLE`	Boldface text
`FL_ITALIC_STYLE`	Guess what
`FL_BOLDITALIC_STYLE`	BoldItalic
`FL_FIXED_STYLE`	Fixed width (good for tables)
`FL_FIXEDBOLD_STYLE`	

`FL_FIXEDITALIC_STYLE`	
`FL_FIXEDBOLDITALIC_STYLE`	
`FL_TIMES_STYLE`	Times-Roman like font
`FL_TIMESBOLD_STYLE`	
`FL_TIMESITALIC_STYLE`	
`FL_TIMESBOLDITALIC_STYLE`	
`FL_SHADOW_STYLE`	Text casting a shadow
`FL_ENGRAVED_STYLE`	Text engraved into the form
`FL_EMBOSSED_STYLE`	Text standing out

The last three styles are special in that they are modifiers, i.e., they do not cause font changes themselves, they only modify the appearance of the font already active. E.g., to get a bold engraved text, set `lstyle` to `FL_BOLD_STYLE|FL_ENGRAVED_STYLE`.

Other styles correspond to the first 12 fonts. The package, however, can handle up to 32 different fonts. The first 16 (numbers 0–15) have been pre-defined. The following table gives their names:

0	helvetica-medium-r
1	helvetica-bold-r
2	helvetica-medium-o
3	helvetica-bold-o
4	courier-medium-r
5	courier-bold-r
6	courier-medium-o
7	courier-bold-o
8	times-medium-r
9	times-bold-r
10	times-medium-o
11	times-bold-o
12	charter-medium-r
13	charter-bold-r
14	charter-medium-i
15	Symbol

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 (helvetica-medium) it should change font `FL_NORMAL_STYLE` To change a font use the following routine:

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

`style` is the number of the font (between 0 and 31) and `name` must contain a valid font name (as known to the X server or font server). If a font name in XLFD is given, a question mark (?) in the point size position informs the **Forms Library** that variable size will be requested later. It is preferable that the complete XLFD name (i.e., with 14 dashes and possibly wildcards) be given because a complete name has the additional advantage that the font may be re-scalable if scalable fonts are available. This means that although both

```
"-*-helvetica-medium-r-*-*-*-?-*-*-*-*-*-*"
"-*-helvetica-medium-r-*-*-*-?-*-*"
```

are valid font names, the first form may be re-scalable while the the second is not.

To obtain the actual built-in font names, use the following function

```
int fl_enumerate_fonts(void (*cb)(const char *f), int shortform)
```

where `cb` is a callback function that gets called once for every built-in font name. The font name is passed to the callback function as the string pointer parameter. `sform` selects if a short form of the name should be used.

XForms only specifies the absolutely needed parts of the font names, and assumes the font path is set so that the server always chooses the most optimal fonts for the system. If this is not true, you can use `fl_set_font_name` to select the exact font you want. In general, this is not recommended if your application is to be run/displayed on different servers.

See `fonts.c` for a demonstration of all the built-in font styles available.

You can 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_lalign(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`	To the top of the box.
`FL_ALIGN_BOTTOM`	To the bottom of the box.
`FL_ALIGN_CENTER`	In the middle of the box.
`FL_ALIGN_RIGHT_BOTTOM`	To the right and bottom of the box.
`FL_ALIGN_LEFT_BOTTOM`	To the left and bottom of the box.
`FL_ALIGN_RIGHT_TOP`	To the right and top of the box.
`FL_ALIGN_LEFT_TOP`	To the left and top of the box.

Normally, all the alignment request places the text outside the box, except for `FL_ALIGN_CENTER`. This can be changed by using a special mask, `FL_ALIGN_INSIDE`, to request alignments that place the text inside the box. This works for most of the objects in the library but not for all. For sliders, inputs and some others, placing the label inside the box simply does not make sense. In these cases, inside request is ignored. See the demo program `lalign.c`for an example use of `FL_ALIGN_INSIDE`.

Finally, the routine

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

changes the label of a given object. The passed parameter `label` is copied internally. As mentioned earlier, newline (`\n`) can be embedded in the label to generate multiple lines. By embedding <CNTRL> `Control` (`\010`) in the label, the entire label or one of the characters in the label can be underlined.

3.11.4 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 an absolute minimum.

3.11.5 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. In addition to the flickering, it is also time consuming. Thus it is more efficient 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, all changes made are instead buffered internally. 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 practice to place multiple changes to the contents of a form always between calls to these two procedures. Further, it is better to complete modifying the attributes of one object before starting work on the next.

3.12 Symbols

Rather than textual labels, it is possible to place symbols like arrows etc. 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 pre-defined symbols are available:

`->`	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.
`->|`	A normal arrow with a bar at the end
`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).
`UpLine`	An embossed line.
`DnLine`	An engraved line.

See Fig. 3.6 for how they look. See also `demo25.c`.

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 of 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 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 draw an arrow at an angle of 30 degrees use 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-directions will be different. To avoid this, i.e., to keep the scaling the same in both directions, put a sharp (`#`) immediately after 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(const 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` is reserved and currently has no meaning. Simply setting it to zero would do.

The routine `drawit` should have the form

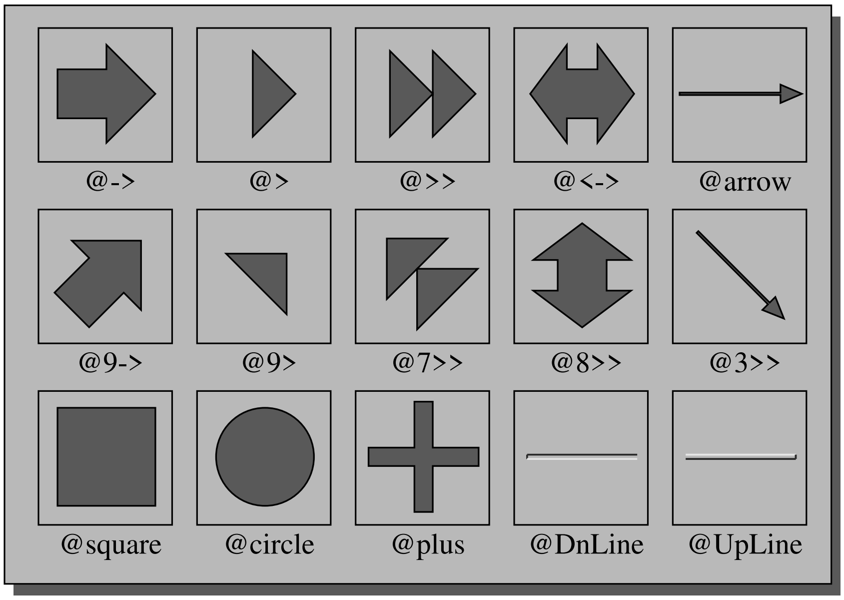

Figure 3.6: Some built-in and rotated symbols

```
void drawit(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
            int angle, FL_COLOR col)
```

`col` is the color in which to draw the symbol. This is the label color that can be provided and changed by the application program. The routine should draw the symbol centered inside the box given by `x`,`y`,`w`,`h` and rotated from its nature position by `angle` degrees.

If the new symbol name is the same as one of the built-ins, the new definition overrides the build-in. Note the the new symbol does not have to be vector graphics, you can use pixmap or whatever in the drawing function.

3.13 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 start adding objects to the form using `fl_add_button` etc. 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 this is not always a good idea because not everything behaves well (e.g. strange things might happen when a group is started but not yet finished).

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. adding the same object to multiple forms will most likely result in a memory fault.) Also watch out with deleting group objects. Not the whole group is deleted, only the object that marks its start is, which gives strange effects.

3.14 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. A freed form should not be referenced.

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

```
Window fl_show_form(FL_FORM *form,int place,int border, const char *name)
```

It opens a (top-level) window on the screen in which the form is shown. The routine returns the window resource ID of the form. You normally never need this. Immediately after the form becomes visible, a full draw of all objects on the form is performed. Due to the two way buffering mechanism of Xlib, if `fl_show_form()` is followed by something that blocks (e.g., waiting for a device other than X devices to come online), the output buffer might not be properly flushed, resulting in the form only being partially drawn. If your program works this way, use `XFlush(fl_get_display())` after `fl_show_form()`. For typical programs that use `fl_do/check_forms()` after `fl_show_form()`, flushing is not necessary.

The location and size of the window are determined by `place`. The following possibilities exist:

`FL_PLACE_SIZE` The user can control the position but the size is fixed. Interactive resizing is not allowed once the form becomes visible.

`FL_PLACE_POSITION` Initial position used will be the one set via `fl_set_form_position()`. Interactive resizing is possible.

`FL_PLACE_GEOMETRY` Place at the latest position and size (see also below) or the geometry set via `fl_set_form_geometry()`. A form so shown will have a fixed size and interactive resizing is not allowed.

`FL_PLACE_ASPECT` Allows interactive resizing but any new size will have the aspect ratio as that of the initial size.

`FL_PLACE_MOUSE` The form is placed centered below the mouse. Interactive resizing will not be allowed unless this option is accompanied by `FL_FREE_SIZE` as in `FL_PLACE_MOUSE|FL_FREE_SIZE`.

`FL_PLACE_CENTER` The form is placed in the center of the screen. If `FL_FREE_SIZE` is also specified, interactive resizing will be allowed.

`FL_PLACE_FULLSCREEN` The form is scaled to cover the full screen. If `FL_FREE_SIZE` is also specified, inter ative resizing will be allowed.

`FL_PLACE_FREE` Both the position and size are completely free. The initial size used is the designed size. Initial position, if set via `fl_set_form_position()`, will be used otherwise inter ative positioning may be possible if the window manager allows it.

`FL_PLACE_HOTSPOT` The form is so placed that mouse is on the "hotspot". If `FL_FREE_SIZE` is also specified, inter ative resizing will be allowed.

`FL_PLACE_CENTERFREE` Same as `FL_PLACE_CENTER|FL_FREE_SIZE`, i.e., place the form at the center of the screen and allow resizing.

`FL_PLACE_ICONIC` The form is shown initially iconified. The size and location used are the window manager's default.

To change the size or location of the icon for `FL_PLACE_ICONIC`, `fl_show_form()` must be done in three steps. First use the following routine

```
Window fl_prepare_form_window(FL_FORM *form)
```

to get the window ID of the form and then use Xlib calls to set icon attributes. After this is done, you must use the following

```
void fl_show_form_window(FL_FORM *form)
```

to make the form visible as an icon having the new attributes.

If size is not specified, the designed (or later scaled) size will be used.

Note that the initial position is dependent upon the window manager used. Some window managers will allow interactive placement of the windows and some will not.

You can set the position or size to be used via the following calls

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

and

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

or more conveniently

```
void fl_set_form_geometry(FL_FORM form*, FL_Coord x, FL_Coord y,
                          FL_Coord w, FL_Coord 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. The position is measured from the top-left corner of the screen. When the position is negative the distance from the right or the bottom is indicated. Next the form should be placed on the screen using `FL_PLACE_GEOMETRY`, `FL_PLACE_FREE`. E.g., to place a form at the lower-right corner of the screen use

```
fl_set_form_position(form,-form->w,-form->h);
fl_show_form(form,FL_PLACE_GEOMETRY,FL_TRANSIENT,"formName");
```

To show a form so that a particular object or point is on the mouse, use one of the following two routines to set the "hotspot"

```
void fl_set_form_hotspot(FL_FORM *form, FL_Coord x, FL_Coord y);

void fl_set_form_hotobject(FL_FORM *form, FL_OBJECT *ob);
```

and use `FL_PLACE_HOTSPOT` in `fl_show_form` to realize. The coordinates `x` and `y` are relative to the upper-left corner of the form.

In the call `fl_show_form()` the argument `border` indicates whether or not to request window manager's decoration. `border` should take one of the following values:

`FL_FULLBORDER`	full border.
`FL_TRANSIENT`	border with (possibly) less decoration.
`FL_NOBORDER`	no decoration at all.

For some dialogs, such as demanding an answer etc., you probably do not want window manager's full decoration. Use `FL_TRANSIENT` for this.

A window border is useful to let the user iconify a form or move it around. If a form is transient or has no border, it is normally more difficult (or even impossible) to move the form. A transient form typically *should* have less decoration, but not necessarily so. It depends on window mangers as well as their options. `FL_NOBORDER` is guaranteed to have no border[1] and is immune to iconification request. Because of this, borderless forms can be hostile to other applications[2], so use this only if absolutely necessary.

There are other subtle differences between the different decoration requests. For instance, (small) transient forms always have `save_under` (See *XSetWindowAttributes*(3X11)) set to true by default. Some window properties, `WM_COMMAND` in particular, are only set for full-bordered forms and will only migrate to other full-bordered forms when the original form having the property becomes unmapped.

The library has a notion of a "main form" of an application, roughly the form that would be on the screen the longest. By default, the first full-bordered form shown becomes the main form of the application. All transient windows shown afterwards will stay on top of the main form. The application can set or change the main form anytime using the following routine

[1] provided the window manager is compliant. If the window manager is not compliant, all bets are off.

[2] Actually, they are also hostile to their sibling forms. See Appendix A

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

Setting the main form of an application will cause the `WM_COMMAND` property set for the form if no other form has this property.

Sometimes, it is necessary to have access to the window resource ID before the window is mapped (shown). For this, the following routine can be used

```
Window fl_prepare_form_window(FL_FORM *form, int place,
                              int border, const char *name)
```

This routine creates a window that obeys any and all constraints just as `fl_show_form` does but remains unmapped. To map such a window, the following must be used

```
Window fl_show_form_window(FL_FORM *form)
```

Between these two calls, the application program has full access to the window and can set all attributes, such as icon pixmaps etc., that are not set by `fl_show_form()`.

You can also scale the form and all objects on it programmatically using the following routine

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

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

When a form is scaled, either programmatically or interactively, all objects on the form will also be scaled. This includes both the size and position of the object. For most cases, this default behavior is adequate. In some cases, e.g., keeping a group of objects together, more control is needed. To this end, the following routines can be used

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

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

where `howresize` can be one of `FL_RESIZE_NONE`, `FL_RESIZE_X` or `FL_RESIZE_Y` with obvious meanings. An alias `FL_RESIZE_ALL`, defined to be `FL_RESIZE_X|FL_RESIZE_Y`, can be used to make both dimension scalable.

`NWgravity` and `SEgravity` control respectively the positioning of the upper-left and lower-right corner of the object and work analogously to the `win_gravity` in Xlib. The details are as follows: Let P be the corner the gravity applies to, (dx1,dy1) the distance to the upper-left corner of the form, (dx2,dy2) the distance to the lower-right corner of the form, then,

Value	Effect
FL_NoGravity	Default linear scaling. See below
FL_NorthWest	dx1, dy1 constant
FL_North	dy1 constant
FL_NorthEast	dy1, dx2 constant
FL_West	dx1 constant
FL_East	dx2 constant
FL_SouthWest	dx1, dy2 constant
FL_South	dy2 constant
FL_SouthEast	dx2, dy2 constant

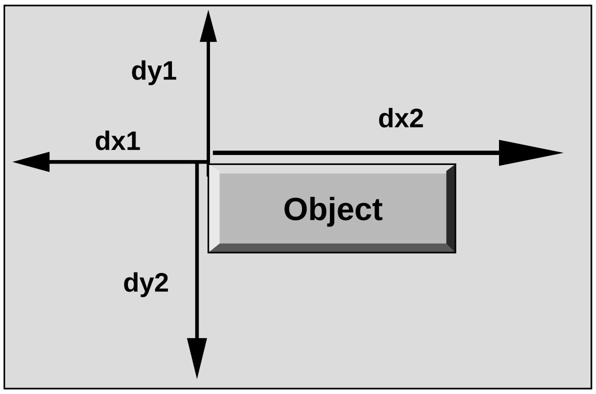

Figure 4.1: Object gravity (NWgravity Shown)

Default for all object is `FL_RESIZE_ALL` and `ForgetGravity`. Note that the three parameters are not orthogonal and the positioning request will always override the scaling request in case of conflict. This means `FL_RESIZE` is consulted only if one (or both) of the gravities is `FL_NoGravity`.

For the special case where `howresize` is `FL_RESIZE_NONE` and both gravities are set to `ForegetGravity`, the object is left un-scaled, but the object is moved so that the new position keeps the center of gravity of the object constant relative to the form.

Again, since all sizing requests go though the window manager, there is no guarantee that your request will be honored. If a form is placed with `Fl_PLACE_GEOMETRY` or other size-restricting options, resizing later via `fl_set_form_size` will likely be rejected.

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. In general, the visual chosen by **XForms** is the one that has the most colors. Application programs have many ways to change this default, either through command line options, resources or programmatically. See the appendices for details.

If for any reason, you would like to change the form title (as well as its associted icon) after it is shown, the following call can be used

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

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

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

where `icon` and `mask` can be any valid Pixmap ID. (See Sections 13.4 and 13.5 for some of the routines that can be used to create Pixmaps) Note that the previous icon, if exists, if not freed or modified in anyway. See demo program `iconify.c` for an example.

If the application program wants to stop interaction with a form and remove it from the screen, 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 deallocate its memory using the call `fl_free_form()` described earlier.

Window managers typically have a menu entry labeled "delete" or "close" meant to terminate an application program gently by informing the application program with a `WM_DELETE_WINDOW` protocol message. Although the **Forms Library** catches this event, it does not do anything except terminating the application. This can cause problems if the application has to do some record keeping before exiting. To perform any record keeping or elect to ignore this message, register a callback function using the following routine

```
fl_set_atclose(int (*at_close)(FL_FORM *, void *), void *data)
```

The callback function `at_close` will be called before the **Forms Library** terminates the application. The first parameter of the callback function is the form that received the `WM_DELETE_WINDOW` message. To prevent the **Forms Library** from terminating the application, the callback function should return a constant `FL_IGNORE`. Any other value (e.g.,`FL_OK`, `FL_IGNORE + 1` etc.) will result in the termination of the application.

Similar mechanism exists for individual forms

```
fl_set_form_atclose(FL_FORM *,
                    int (*at_close)(FL_FORM *, void *),
                    void *data)
```

except that `FL_OK` does not terminate the application, it results in the form being closed.

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, appropriate for simple programs, 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. `touchbutton.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 some of the demo programs for examples of 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 Non-blocking interaction

The simple interaction mentioned above is adequate 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 time a `NULL` object is returned. Programs working in this way have the following as the 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. It is possible to achieve the same effect without using `fl_check_forms()` by the means of callbacks (`fl_set_idle_callback()`), which is the prefered method of interaction for reasonably complicated programs.

4.3 Dealing with multiple windows

It is not atypical that an application program may need to take interaction from more than one form at the same time, **Forms Library** provides a mechanism with which precise control can be exercised.

By default, `fl_do_forms()` takes interaction from all forms that are shown. In certain situations, you might not want to have interaction with all of them. For example, 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. The user first has to give the confirmation. So you want to temporarily 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 a good idea to give the user a visual clue that a form is deactivated. This is not automatically done mainly for performance reasons. Experience shows that graying out some important objects on the form is in general adequate. Graying out an object can be accomplished by using `fl_set_object_lcol()` (See `objinactive.c`). What objects to gray out is obviously application dependent.

The following two functions can be used to register two callbacks that that are called whenever the activation status of a form is changed:

```
typedef void (*FL_FORM_ATACTIVATE)(FL_FORM *, void *);

FL_FORM_ATACTIVATE fl_set_form_atactivate(FL_FORM *form,
          FL_FORM_ATACTIVATE callback, void *data);

typedef void (*FL_FORM_ATDEACTIVATE)(FL_FORM *, void *);
FL_FORM_ATDEACTIVATE fl_set_form_atdeactivate(FL_FORM *form,
          FL_FORM_ATDEACTIVATE callback, void *data);
```

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 not read the event queue while displaying forms. To solve this problem, the package maintains (or appears to maintain) 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 `Xlib` routines. Instead of using `XNextEvent()`, the program should use `fl_XNextEvent()`, with the same parameters except the `Display *`. The following is a list of all new routines:

```
int fl_XNextEvent(XEvent *xev);
int fl_XPeekEvent(XEvent *xev);
int fl_XEventsQueued(int mode);
int fl_XPutbackEvent(XEvent *xev);
```

Other events routines may be directly used if proper care is taken to make sure that only events for the application window in question are removed. These routines include `XWindowEvent`, `XCheckWindowEvent` etc.

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`. Upon receiving this special event, the application program can and *must* remove the pending event from the queue using `fl_XNextEvent()`.

So the basis of a program with its own windows would look as follows:

```
/* define the forms */
/* display the forms */
/* open your own window(s) */
while (! ready)
{
    obj = fl_do_forms(); /* or fl_check_forms() */
    if (obj == FL_EVENT)
    {
       fl_XNextEvent(&xevent);
       switch(xevent.type)
       {
       /* 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 function that pops up a form to change some settings. This routine might not want to be concerned with any redrawing of the main window, etc., but you also don't want to discard any events. 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()`.

It can't be over-emphasized that it is an error to ignore `FL_EVENT` or use `fl_XNextEvent()` without seeing `FL_EVENT`.

Sometimes an application program might need to find out what event triggered a callback, e.g., to implement button number sensitive functionalities. To this end, the following routine may be called

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

Note that if this routine is used outside of a callback function, the value returned may not be the real "last event" if the program was idling and in this case, it returns a synthetic `MotionNotify` event.

Some of the utilities used internally by the **Forms Library** can be used by the application programs, such as window geometry queries etc. Following is a partial list of the available routines:

```
void fl_get_winorigin(Window win, FL_Coord *x, FL_Coord *y);

void fl_get_winsize(Window win, FL_Coord *w, FL_Coord *h);

void fl_get_wingeometry(Window win, FL_Coord *x, FL_Coord *y,
                         FL_Coord *w, FL_Coord *h);
```

All positions are relative to the root window.

There are also routines that can be used to obtain the current mouse position relative to the root window:

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

where `keymask` is the same as used in *XQueryPointer(3X11)*. The function returns the window ID the mouse is in.

To obtain the mouse position relative to an arbitrary window, the following routine may be used

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

To print the name of an XEvent, the following routine can be used:

```
XEvent *fl_print_xevent_name(const char *where, const XEvent *xev)
```

The function takes an XEvent, prints out its name and some other info, e.g., `expose, count=n`. Parameter `where` can be used to indicate where this function is called:

```
fl_print_xevent_name("In tricky.c",&xevent);
```

4.4 Using callback functions

As stated earlier, the recommended method of interaction is to use callback functions. A callback function is a function supplied to the library by the application program that binds a specific condition (e.g., a button is pushed) to the invocation of the function by the system.

The application program can bind a callback routine to any object. Once a callback function is bound and the specified condition is met, `fl_do_forms()` or `fl_check_forms()` invokes the callback function instead of returning the object.

To bind a callback routine to an object, use the following

```
typedef void (*FL_CALLBACKPTR)(FL_OBJECT *ob, long argument);
FL_CALLBACKPTR fl_set_object_callback(FL_OBJECT *obj,
                                      FL_CALLBACKPTR callback,
                                      long argument)
```

where `callback` is the callback function. `argument` is an argument that is passed to the callback routine so that it can take different actions for different objects. Function returns the old callback routine already bound to the object. You can change the callback routine anytime using this function. See, for example, demo program `timer.c`.

The callback routine should have the form

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

The first argument to every callback function is the object to which the callback is bound. The second parameter is the argument specified by the application program in the call to `fl_set_object_callback()`.

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

Sometimes it is necessary to access other objects on the form from within the callback function. This presents a difficult situation that calls for global variables for all the objects on the form. This runs against good programming methodology and can make a program hard to maintain. **Forms Library** solves (to some degree) this problem by creating two fields, `void *u_vdata` and `long u_ldata`, in the `FL_OBJECT` structure that you can use to hold the necessary data to be used in the callback function. Better and more general solution to the problem is detailed in Part II of this documentation where all objects on a form is grouped into a single structure which can then be “hang" off of `u_vdata`.

Another communication problem might arise when the callback function is called and from within the callback function, some other objects' state is explicitly changed, say, via `fl_set_button`,`fl_set_input` etc. You probably don't want to put the state change handling code of these objects in another object's callback. To handle this situation, you can simply call

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

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

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

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, void *data)
```

With each form you can associate its own callback routine.

When the application program also has its own windows, it most likely also wants to know about XEvent for the window. As explained earlier, this can be accomplished by checking for `FL_EVENT` objects. Another way is to add an event callback routine. This routine will be called whenever an XEvent is pending for the application's own window. To setup an event callback routine use the call

```
FL_APPEVENT_CB
fl_set_event_callback(int (*callback)(XEvent *ev, void *data),
                     void *data)
```

Whenever an event takes place callback is called with the event as argument. So callback should have the form

```
typedef int (*FL_APPEVENT_CB)(XEvent *ev, void *data);
int callback(XEvent *xev, void *data);
```

This assumes the application program solicits the events and further, the callback routine should be prepared to handle all XEvent for all non-form windows. This could be undesirable if more than one application window is active. To further partition and simplify the interaction, callbacks for a specific event on a specific window can be registered:

```
FL_APPEVENT_CB fl_add_event_callback(Window window, int xev_type,
                          FL_APPEVENT_CB callback, void *user_data);
```

Where window is the window for which the callback routine is to be registered. xev_type is the XEvent type you're interested in, e.g., Expose etc. If xev_type is 0, it is taken to mean the callback routine will handle all events for the window. The newly installed callback replaces the callback already installed.

fl_add_event_callback() does not alter the window's event mask nor solicit events for you. This is so mainly for the reason that an event type does not always correspond to a unique event mask, also in this way, the user can solicit events at window's creation and use 0 to register all the event handlers.

To let **XForms** handle solicitation for you, call the following routine

```
void fl_activate_event_callbacks(Window win)
```

before mapping the window. Note however, this sometimes might not be what you want. For example, MotionNotify event can be solicited using either ButtonMotionMask or PointerMotionMask and **Forms Library** will use both.

It is possible to add or remove solicited event masks on the fly without altering other masks already selected:

```
long fl_addto_selected_xevent(Window win, long mask)

long fl_remove_selected_xevent(Window win, long mask)
```

Both functions return the resulting event masks that are currently selected.

A special callback function, idle_callback can be registered:

```
FL_APPEVENT_CB
fl_set_idle_callback(FL_APPEVENT_CB callback, void *user_data)
```

The callback function is called whenever there are no events pending for forms or application windows. This setup can be useful if you need to keep working on some non-CPU intensive tasks periodically or checking and servicing some external devices. It is not appropriate for tasks that block or take a long time to finish.

The callback function will be called as follows

```
int callback(xev, user_data);
```

Where `xev` is a pointer to a synthetic[3] `MotionNotify` event from which mouse position etc. can be obtained.

If event callback functions are registered via both `fl_set_event_callback()` and `fl_add_event_callback()`, the callback via the latter is invoked first and the callback registered via `fl_set_event_callback()` is called only if the first attempt is unsuccessful, that is, the handler for the event is not present. For example, after the following sequence

```
fl_add_event_callback(WinID, Expose, expose_cb, 0);
fl_set_event_callback(event_callback);
```

All `Expose` events on window `WinID` are consumed by `expose_cb`, thus `event_callback` would never be invoked as a result of an `Expose` event.

To remove a callback, use the following routine

```
void fl_remove_event_callback(Window win, int xev_type)
```

All parameters have the usual meaning. Again, this routine does not modify the window's event mask. If you like to change the events the window is sensitive to after removing the callback, use `fl_activate_event_callbacks()`. If `xev_type` is 0, all callbacks for window `win` are removed. This routine is called automatically if `fl_winclose()` is called to unmap and destroy a window. Otherwise, you must call this routine explicitly to remove all event callbacks before destroying a window using `XDestroyWindow()`. .

A program using all of these has the following basic form:

```
void event_cb(XEvent *xev, void *mydata1)
{ /* Handles an X-event. */ }

void expose_cb(XEvent *xev, void *mydata2)
{ /* handle expose */ }

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

void form2_cb(FL_OBJECT *obj)
```

[3] I.e., `xev->xmotion.send_event` is true

```
{ /* Handles object obj in form2. */ }

main(int argc, char *argv[])
{
   /* initialize             */
   /* create form1 and form2 and display them */
   fl_set_form_callback(form1,form1cb);
   fl_set_form_callback(form2,form2cb);
   /* create your own window, winID and show it */
   XSelectInput(fl_get_display(), winID, ExposureMask|ButtonPressMask|...)
   XMapRaised(fl_get_display(),winID);
   fl_set_event_callback(event_cb, whatever);
   fl_add_event_callback(winID,Expose, expose_cb, data);
   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 callback routines (either for the objects or for entire forms). This ensures that no events are missed, events are treated in the correct order, etc. Note that different event callback routines can be written for different stages of the program and they can be switched when required. This provides a progressive path for building up programs.

Another possibility is to use a free object so that the application window is handled automatically by the internal event processing mechanism just like any other forms.

4.5 Handling other input sources

It is not uncommon that X applications may require input from sources other than the X event queue. Outlined in this section are two routines in the **Forms Library** that provide a simple interface to handle additional input sources. Applications can define input callbacks to be invoked when input is available from a specified file descriptor.

The function

```
typedef void (*FL_IO_CALLBACK)(int fd, void *data)
void fl_add_io_callback(int fd, unsigned condition,
                        FL_IO_CALLBACK callback, void *data)
```

registers an input callback with the system. The argument `fd` must be a valid file descriptor on a UNIX-based system or other operating system dependent device specification while `condition` indicates under what circumstance the input callback should be invoked. The `condition` must be one of the following constants

`FL_READ`	File descriptor has data available.
`FL_WRITE`	File descriptor is available for writing.
`FL_EXCEPT`	an I/O error has occured.

When the given condition occurs, the **Forms Library** invokes the callback function specified by `callback`. The `data` argument allows the application to provide some data to be passed to the callback function when it is called (be sure that the storage pointed to by `data` has global (or static) scope).

To remove a callback that is no longer needed or to stop the **Forms Library**'s main loop from watching the file descriptor, use the following function

```
void fl_remove_io_callback(int fd, unsigned condition,
                           FL_IO_CALLBACK callback)
```

The procedures outlined above work well with pipes and sockets, but can be a CPU hog on real files. To workaround this problem, you may wish to check the file periodically and only from within an idle callback.

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 IV of this document.) This is especially useful when your new type of objects is of general use.

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. This chapter will give all the details needed to design and use of free objects.

5.1 Free object

To add a free object to a form use the call

```
FL_OBJECT *fl_add_free(int type,FL_Coord x,FL_Coord y,
          FL_Coord w,FL_Coord h, const 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, FL_Coord mx, FL_Coord my,
           int key, void *xev)
```

where `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 meaningful with mouse related events) relative to the form origin and `key` is the KeySym of the key typed in by the user (only for `FL_KEYBOARD` events). `xev` is the (cast) XEvent that causes the invocation of this handler. `event` and `xev->type` can both be used to obtain the event types. 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->x, obj->y`, `obj->w` and `obj->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. This can be figured out as follows. The field `obj->belowmouse` indicates whether the object is below the mouse. The field `obj->pushed` indicates whether the object is currently being pushed with the mouse. Finally, `obj->focus` indicates whether input focus is directed towards this object. When required, the label should also be drawn. This label can be found in the field `obj->label`. The drawing should be done such that it works correctly in the visual/depth the current form is in. Complete information is available on the state of the current form as well as several routines that will help you to tackle the trickiest (also the most tedious) part of X programming. In particular, the return value of `fl_get_vclass()` can be used as an index into a table of structures, `FL_STATE fl_state[]`, from which all information about current active visual can be obtained.

See chapter 23 for details on drawing objects and the routines.

`FL_DRAWLABEL` This event is not always generated. It typically follows `FL_DRAW` and indicates the object label needs to be (re)drawn. You can ignore this event if (a) the object handler always draws the label upon receiving `FL_DRAW` or (b) the object label is not drawn at all[1].

`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 an `FL_DRAW` event to the object but also does some other things (like setting window id's, taking care of double buffering and some other bookkeeping tasks).

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

`FL_MOTION` A motion event is sent between `FL_ENTER` and `FL_LEAVE` events when the mouse position changes on the object. The mouse position is given with the routine.

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

`FL_RELEASE` The user has released the mouse button. This event is only sent if a PUSH event was sent earlier.

`FL_DBLCLICK` The user has pushed a mouse button twice within a certain time limit (`FL_CLICK_TIMEOUT`).

[1] Label for free objects can't be drawn outside of the bounding box because of the clippings by the dispatcher

FL_TRPLCLICK The user has pushed a mouse button three times within a certain time window between each push. This event is sent after a FL_DBLCLICK, FL_PUSH, FL_RELEASE sequence.

FL_MOUSE The mouse position has changed. This event is sent to an object between an FL_PUSH and an FL_RELEASE event. The mouse position is given as the parameter mx and my and action can be taken based on the position.

FL_FOCUS Input got focussed to this object. This event and the next two are only sent 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 KeySym is given with the routine. This event only happens between FL_FOCUS and FL_UNFOCUS events.

FL_STEP A step event is sent all the time (at most 50 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.

FL_SHORTCUT Hotkeys for the object have been triggered. Typically this should result in the returning of the free object.

FL_FREEMEM Upon receiving this event, the handler should free all object class specific memory allocated.

FL_OTHER Some other events typically caused by window manager events or inter-client events. All information regarding the details of the events is in xev.

Many of these events might make it necessary to (partially) redraw the object. Always do this using the routine fl_redraw_object().

As indicated above not all events are sent to all free objects. It depends on their types. The following types exist (all objects are sent FL_OTHER when it occurs):

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

FL_INACTIVE_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 events.

See free1.c for a (terrible) example of the use of free objects. See also freedraw.c, which is a nicer example of the use of free objects.

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

5.2 An Example

We conclude our discussion of the free object by examining a simple drawing program capable of drawing simple geometric figures like squares, circles, and triangles of various colors and sizes, and of course it also utilizes a free object.

The basic UI consists of three logical parts. A drawing area onto which the squares etc. are to be drawn; a group of objects that control what figure to draw and with what size; and a group of objects that control the color with which the figure is to be drawn.

The entire UI (see Fig. 5.1) is designed interactively using the GUI builder `fdesign` with most objects having their own callbacks. `fdesign` writes two files, one is a header file containing forward declarations of callback functions and other function prototypes:

```
#ifndef FD_drawfree_h_
#define FD_drawfree_h_

extern void change_color(FL_OBJECT *, long);
extern void switch_figure(FL_OBJECT *, long);
/* more callback declarations omitted */

typedef struct {
     FL_FORM *drawfree;
     FL_OBJECT *freeobj;
     FL_OBJECT *figgrp;
     FL_OBJECT *colgrp;
     FL_OBJECT *colorobj;
     FL_OBJECT *rsli;
     FL_OBJECT *gsli;
     FL_OBJECT *bsli;
     FL_OBJECT *miscgrp;
     FL_OBJECT *sizegrp;
     FL_OBJECT *hsli;
     FL_OBJECT *wsli;
     FL_OBJECT *drobj[3];
     void *vdata;
     long ldata;
} FD_drawfree;

extern FD_drawfree *create_form_drawfree(void);
#endif /* FD_drawfree_h_ */
```

The other file contains the actual C-code that creates the form when compiled and executed. Since free object is not directly supported by `fdesign`, a box was used as a stub for the location and size of the drawing area. After the C-code was generated, the box was changed manually to a free object by replacing `fl_add_box(FL_DOWN_BOX ...)` with `fl_add_free(FL_NORMAL_FREE,...)`. We list below the output generated by `fdesign` with some comments:

```
FD_drawfree *create_form_drawfree(void)
{
  FL_OBJECT *obj;
  FD_drawfree *fdui = (FL_drawfree *)fl_calloc(1, sizeof(FD_drawfree));

  fdui->drawfree = fl_bgn_form(FL_NO_BOX, 530, 490);
  obj = fl_add_box(FL_UP_BOX,0,0,530,490,"");
```

This is almost always the same for any form definition: we allocate a structure that will hold all objects on the form as well as the form itself. In this case, the first object on the form is a box of type `FL_UP_BOX`.

```
  fdui->figgrp = fl_bgn_group();
  obj = fl_add_button(FL_RADIO_BUTTON,10,60,40,40,"@#circle");
    fl_set_object_lcol(obj,FL_YELLOW);
    fl_set_object_callback(obj,switch_figure,0);
  obj = fl_add_button(FL_RADIO_BUTTON,50,60,40,40,"@#square");
    fl_set_object_lcol(obj,FL_YELLOW);
    fl_set_object_callback(obj,switch_figure,1);
  obj = fl_add_button(FL_RADIO_BUTTON,90,60,40,40,"@#8>");
    fl_set_object_lcol(obj,FL_YELLOW);
    fl_set_object_callback(obj,switch_figure,2);
  fl_end_group();
```

This creates three buttons that control what figures are to be drawn. Since figure selection is mutually exclusive, we use `RADIO_BUTTON` for this. Further, the three buttons are placed inside a group so that they won't interfere with other radio buttons on the same form. Notice that callback function `switch_figure()` is bound to all three buttons but with different arguments. The callback function can resolve the associated object with the callback function argument. In this case, 0 is used for circle, 1 for square and 2 for triangle. This association of a callback function with a piece of user data can often reduce coding substantially, especially if you have a large group of objects that control similar things. The advantage will become clear as we proceed.

Next we add three sliders to the form. By using appropriate colors for the sliding bar (Red, Green, Blue), there is no need to label the slider.

```
  fdui->colgrp = fl_bgn_group();
  fdui->rsli=obj=fl_add_slider(FL_VERT_FILL_SLIDER,25,170,30,125,"");
    fl_set_object_color(obj,FL_COL1,FL_RED);
    fl_set_object_callback(obj,change_color,0);
  fdui->gsli=obj=fl_add_slider(FL_VERT_FILL_SLIDER,55,170,30,125,"");
```

```
  fl_set_object_color(obj,FL_COL1,FL_GREEN);
  fl_set_object_callback(obj,change_color,1);
fdui->bsli=obj=fl_add_slider(FL_VERT_FILL_SLIDER,85,170,30,125,"");
  fl_set_object_color(obj,FL_COL1,FL_BLUE);
  fl_set_object_callback(obj,change_color,2);
fdui->colorobj = obj = fl_add_box(FL_BORDER_BOX,25,140,90,25,"");
  fl_set_object_color(obj,FL_FREE_COL1,FL_FREE_COL1);
fl_end_group();
```

Again, a single callback function, `change_color()`, is bound to all three sliders. In addition to the sliders, a box object is added to the form. This box is set to use color index `FL_FREE_COL1` and will be used to show visually what the current color setting looks like. This implies that in the `change_color()` callback function, the entry `FL_FREE_COL1` in the **Forms Library**'s internal colormap will be changed. We also place all the color related objects inside a group even though they are not of radio property. This is to facilitate gravity settings which otherwise require setting the gravities of each individual object.

Next we create our drawing area which is simply a free object of type `NORMAL_FREE` with a handler to be written

```
obj = fl_add_frame(FL_DOWN_FRAME,145,30,370,405,"");
  fl_set_object_gravity(obj, FL_NorthWest, FL_SouthEast);
fdui->freeobj = obj = fl_add_free(FL_NORMAL_FREE,145,30,370,405,"",
                      freeobject_handler);
  fl_set_object_boxtype(obj, FL_FLAT_BOX);
  fl_set_object_gravity(obj, FL_NorthWest, FL_SouthEast);
```

The frame is added for decoration purpose only. Although a free object with a down box would appear the same, the down box can be written over by the free object drawing while the free object can't draw on top of the frame since the frame is outside of the free object. Notice the gravity settings. This kind setting maximizes the real estate of the free object when the form is resized.

Next, we need to have control over the size of the object. For this, added are two sliders bound to the same callback function with different user data (0 and 1 in this case):

```
fdui->sizegrp = fl_bgn_group();
fdui->wsli=obj=fl_add_valslider(FL_HOR_SLIDER,15,370,120,25,"Width");
  fl_set_object_lalign(obj,FL_ALIGN_TOP);
  fl_set_object_callback(obj,change_size,0);
fdui->hsli=obj=fl_add_valslider(FL_HOR_SLIDER,15,55,410,25,"Height");
  fl_set_object_lalign(obj,FL_ALIGN_TOP);
  fl_set_object_callback(obj,change_size,1);
fl_end_group();
```

The rest of the UI consists of some buttons the user can use to exit the program, elect to draw outline instead of filled figures etc. Form definition ends with `fl_end_form()`. The structure that holds the form as well as all the objects on them is returned to the caller:

```
   fdui->miscgrp = fl_bgn_group();
   obj = fl_add_button(FL_NORMAL_BUTTON,395,445,105,30,"Quit");
     fl_set_button_shortcut(obj,"Qq#q",1);
   obj = fl_add_button(FL_NORMAL_BUTTON,280,445,105,30,"Refresh");
     fl_set_object_callback(obj,refresh_cb,0);
   obj = fl_add_button(FL_NORMAL_BUTTON,165,445,105,30,"Clear");
     fl_set_object_callback(obj,clear_cb,0);
   fl_end_group();

   obj = fl_add_checkbutton(FL_PUSH_BUTTON,15,25,100,35,"Outline");
     fl_set_object_color(obj,FL_MCOL,FL_BLUE);
     fl_set_object_callback(obj,fill_cb,0);
     fl_set_object_gravity(obj,FL_NorthWest,FL_NorthWest);
   fl_end_form();
   return fdui;
```

After creating the UI, we need to write the callback functions and the free object handler. The callback functions are relatively easy since each object is designed to perform a very specific task.

Before we proceed to code the callback functions, we first need to define the overall data structure that will be used to glue together the UI and the routines that do real work.

The basic structure is the `DrawFigure` structure that holds the current drawing function as well as object attributes such as size and color:

```
#define MAX_FIGURES   500
typedef void (*DrawFunc)(int/*fill */, int,int,int,int,/* x,y,w,h */
                        FL_COLOR )    /* color   */
typedef struct
{
   DrawFunc drawit;      /* how to draw this figure */
   int fill, x,y,w,h;    /* geometry                */
   int pc[3];            /* primary color R,G,B     */
   int newfig;           /* indicate a new figure   */
   FL_COLOR col;         /* FL color index          */
} DrawObject;

static DrawFigure saved_figure[MAX_FIGURES], *cur_fig;
static FD_drawfree *drawui;
int max_w = 30, max_h = 30;   /* max size of figures */
```

All changes to the figure attributes will be buffered in `cur_fig` and when the actual drawing command is issued (mouse click inside the free object), `cur_fig` is copied into `saved_figure` array buffer.

Forms Library contains some low-level drawing routines that can draw and optionally fill arbitrary polygonal regions, so in principle, there is no need to use Xlib calls directly. To show how Xlib drawing routine is combined with **Forms Library**, we use Xlib routines to draw a triangle:

```
void draw_triangle(int fill, int x, int y, int w, int h, FL_COLOR col)
{
     XPoint xp[4];
     GC gc = fl_state[fl_get_vclass()].gc[0];
     Window win = fl_winget();
     Display *disp = fl_get_display();

     xp[0].x = x;         xp[0].y = y + h - 1;
     xp[1].x = x + w/2;   xp[1].y = y;
     xp[2].x = x + w - 1; xp[2].y = y + h - 1;
     XSetForeground(disp, gc, fl_get_pixel(col));
     if(fill)
       XFillPolygon (disp, win, gc, xp, 3, Nonconvex, Unsorted);
     else
     {
         xp[3].x = xp[0].x; xp[3].y = xp[0].y;
         XDrawLines(disp, win, gc, xp, 4, CoordModeOrigin);
     }
}
```

Although more or less standard stuff, some explanation is in order. As you have probably guessed, `fl_winget()` returns the current "active" window, defined to be the window the object receiving dispatcher's messages (`FL_DRAW` e.g.) belongs to. Similarly the routine `fl_get_display()` returns the current connection to the X server. Part IV has more details on the utility functions in the **Forms Library**.

The structure `fl_state[]` keeps much "inside" information on the state of the **Forms Library**. For simplicity, we choose to use the **Forms Library**'s default GC. There is no fundamental reason that this has be so. We certainly can copy the default GC and change the foreground color in the copy. Of course unlike using the default GC directly, we might have to set the clip mask in the copy whereas the default GC always have the proper clip mask (in this case, to the bounding box of the free object).

We use the **Forms Library**'s built-in drawing routines to draw circles and rectangles. Then our drawing functions can be defined as follows:

```
static DrawFunc drawfunc[] =
{
   fl_oval, fl_rectangle, draw_triangle
};
```

Switching what figure to draw is just changing the member `drawit` in `cur_fig`. By using the proper object callback argument, figure switching is achieved by the following callback routine that is bound to all figure buttons

```
void switch_object(FL_OBJECT *obj, long which)
{
```

```
        cur_fig->drawit = drawfunc[which];
    }
```

So this takes care of the drawing functions.

Similarly, the color callback function can be written as follows

```
void change_color(FL_OBJECT * ob, long which)
{
   cur_fig->c[which] = fl_get_slider_value(ob) * 255;
   fl_mapcolor(cur_fig->col,cur_fig->c[0],cur_fig->c[1],cur_fig->c[2]);
   fl_mapcolor(FL_FREE_COL1,cur_fig->c[0],cur_fig->c[1],cur_fig->c[2]);
   fl_redraw_object(drawui->colorobj);
}
```

The first `fl_mapcolor` defines the RGB components for index `cur_fig->col` and the second `fl_mapcolor` defines the RGB component for index `FL_FREE_COL1`, which is the color index used by `colorobj` that serves as current color visual feedback.

Object size is taken care of in a similar fashion by using a callback function bound to both size sliders:

```
void change_size(FL_OBJECT * ob, long which)
{
    if (which == 0)
          cur_fig->w = fl_get_slider_value(ob);
    else
          cur_fig->h = fl_get_slider_value(ob);
}
```

Lastly, we toggle the fill/outline option by querying the state of the push button

```
void outline_callback(FL_OBJECT *ob, long data)
{
    cur_fig->fill = !fl_get_button(ob);
}
```

To clear the drawing area and delete all saved figures, a Clear button is provided with the following callback:

```
void clear_cb(FL_OBJECT *obj, long notused)
{
     saved_figure[0] = *cur_fig; /* copy attributes */
     cur_fig = saved_figure;
     fl_redraw_object(drawui->freeobj);
}
```

To clear the drawing area and redraw all saved figures, a Refresh button is provided with the following callback:

```
void refresh_cb(FL_OBJECT *obj, long notused)
{
    fl_redraw_object(drawui->freeobj);
}
```

With all attributes and other services taken care of, it is time to write the free object handler. The user can issue a drawing command inside the free object by clicking either the left or right mouse button.

```
int
freeobject_handler(FL_OBJECT * ob, int event, FL_Coord mx, FL_Coord my,
                  int key, void *xev)
{
    DrawFigure *dr;

    switch (event)
    {
    case FL_DRAW:
        if (cur_fig->newfig == 1)
           cur_fig->drawit(cur_fig->fill,
                           cur_fig->x + ob->x,cur_fig->y + ob->y,
                           cur_fig->w, cur_fig->h, cur_fig->col);
        else
        {
           fl_drw_box(ob->boxtype, ob->x, ob->y, ob->w, ob->h, ob->col1,
                      ob->bw);

           for (dr = saved_figure; dr < cur_fig; dr++)
           {
              fl_mapcolor(FL_FREE_COL1, dr->c[0], dr->c[1], dr->c[2]);
              dr->drawit(dr->fill,dr->x + ob->x,dr->y + ob->y,
                         dr->w,dr->h,dr->col);
           }
        }
        cur_fig->newfig = 0;
        break;
     case FL_PUSH:
        if (key != 2)
        {
           cur_fig->x = mx - cur_fig->w/2;
           cur_fig->y = my - cur_fig->h/2;

           /* convert figure center to relative to the free object*/
           cur_fig->x -= ob->x;
```

```
            cur_fig->y -= ob->y;

            cur_fig->newfig = 1;
            fl_redraw_object(ob);
            *(cur_fig+1) = *cur_fig;
            fl_mapcolor(cur_fig->col+1, cur_fig->c[0], cur_fig->c[1],
                        cur_fig->c[2],
            cur_fig++;
            cur_fig->col++;
         }
         break;
     }
     return 0;
 }
```

In this particular program, we are only interested in mouse clicks and redraw. The event dispatching routine cooks the X event and drives the handler via a set of events (messages). For a mouse click inside the free object, its handler is notified with an `FL_PUSH` together with the current mouse position `mx, my`. In addition, the driver also sets the clipping mask to the bounding box of the free object prior to sending `FL_DRAW`. Mouse position (always relative to the origin of the form) is directly usable in the drawing function. However, it is a good idea to convert the mouse position so it is relative to the origin of the free object if the position is to be used later. The reason for this is that the free object can be resized or moved in ways unknown to the handler and only the position relative to the free object is meaningful in these situations.

It is tempting to call the drawing function in response to `FL_PUSH` since it is `FL_PUSH` that triggers the drawing. However, it is a (common) mistake to do this. The reason is that much bookkeeping is performed prior to sending `FL_DRAW`, such as clipping, double buffer preparation and possibly active window setting etc. All of these is not done if the message is other than `FL_DRAW`. So always use `fl_redraw_object()` to draw unless it is a response to `FL_DRAW`. Internally `fl_redraw_object()` calls the handler with `FL_DRAW` (after some bookkeeping), so we only need to mark `FL_PUSH` with a flag `newfig` and let the drawing part of the handler draw the newly added figure.

`FL_DRAW` has two parts. One is simply to add a figure indicated by `newfig` being true and in this case, we only need to draw the figure that is being added. The other branch might be triggered as a response to damaged drawing area resulting from `Expose` event or as a response to Refresh command. we simply loop over all saved figures and (re)draw each of them.

The only thing left to do is to initialize the program, which includes initial color and size, and initial drawing function. Since we will allow interactive resizing and also some of the objects on the form are not resizeable, we need to take care of the gravities.

```
void draw_initialize(FD_drawfree *ui)
{

   fl_set_form_minsize(ui->drawfree,530,490);
   fl_set_object_gravity(ui->colgrp, FL_West, FL_West);
   fl_set_object_gravity(ui->sizegrp, FL_SouthWest, FL_SouthWest);
```

```
    fl_set_object_gravity(ui->figgrp, FL_NorthWest, FL_NorthWest);
    fl_set_object_gravity(ui->miscgrp, FL_South, FL_South);
    fl_set_object_resize(ui->miscgrp, FL_RESIZE_NONE);

    cur_fig = saved_figure;
    cur_fig->pc[0] = cur_fig->pc[1] = cur_fig->pc[2] = 127;
    cur_fig->w = cur->fig->h = 30;
    cur_fig->drawit = fl_oval;
    cur_fig->col = FL_FREE_COL1 + 1;
    cur_fig->fill = 1;
    fl_set_button(ui->drobj[0], 1); /* show current selection */

    fl_mapcolor(cur_fig->col, cur_fig->pc[0],
                cur->fig->pc[1], cur->fig->pc[2]);
    fl_mapcolor(FL_FREE_COL1, cur_fig->pc[0],
                cur->fig->pc[1], cur->fig->pc[2]);

    fl_set_slider_bounds(ui->wsli, 1, max_w);
    fl_set_slider_bounds(ui->hsli, 1, max_h);
    fl_set_slider_precision(ui->wsli, 0);
    fl_set_slider_precision(ui->hsli, 0);
    fl_set_slider_value(ui->wsli, cur_fig->w);
    fl_set_slider_value(ui->hsli, cur_fig->h);

 }
```

With all the parts in place, the main program simply creates, initializes and shows the UI, then enters the main loop:

```
int main(int argc, char *argv[])
{
    fl_initialize(&argc, argv, "FormDemo", 0, 0);
    drawui = create_form_drawfree();
    draw_initialize(drawui);
    fl_show_form(drawui->drawfree, FL_PLACE_CENTER|FL_FREE_SIZE,
                 FL_FULLBORDER, "Draw");
    fl_do_forms();
    return 0;
}
```

Since the only object that does not have a callback is the Quit button, `fl_do_forms()` will return only if that button is pushed.

Full source code to this simple drawing program can be found in `DEMOS/freedraw.c`.

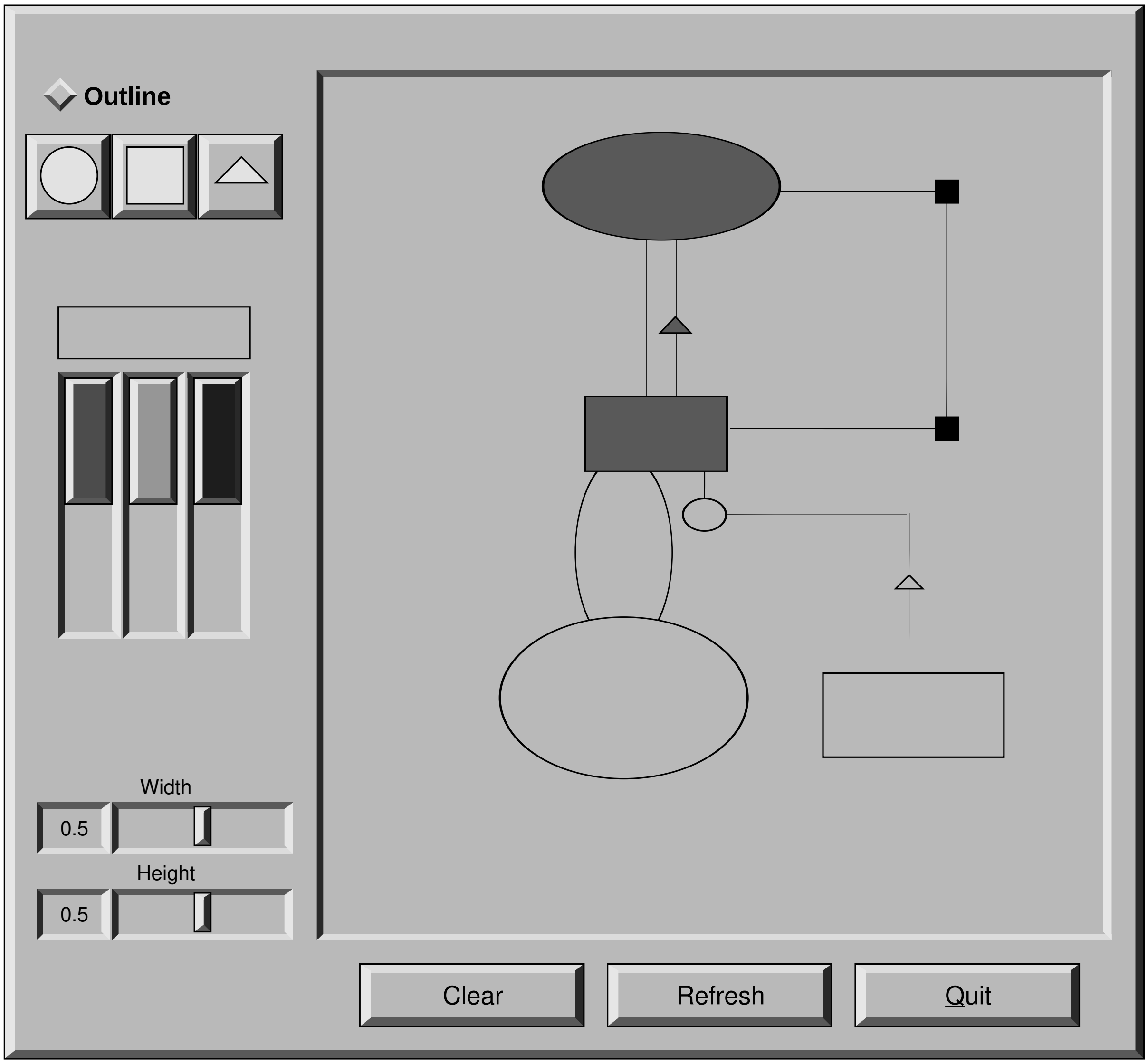

Figure 5.1: Drawing using Free Object

Chapter 6

Goodies

A number of special routines are provided that make working with simple forms even simpler. All these routines build simple forms and handle 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(const char *s1, const char *s2, const char *s3)
```

It shows a simple form with a message and a button with OK on it. `s1`, `s2` and `s3` are the three lines of the message. (Use empty strings if you don't need all the lines.) The routine returns when the user presses the OK button or presses the <RETURN> key.

There is also a routine that can be used to show ar one-line message that can only be removed programmatically

```
void fl_show_oneliner(const char *str, FL_Coord x, FL_Coord y)
void fl_hide_oneliner(void);
```

where `str` is the message and `x` and `y` are the coordinate (relative to the root window) the message should be placed. Note that multi-line message is possible by embedding the newline character in `str`. See the demo program `preemptive.c` for an example of its use.

By default, the background of the message is yellow and the text black. To change this default, use the following routine

```
void fl_set_oneliner_color(FL_COLOR background, FL_COLOR textcol)
```

Similar routine exists to change the font style and size

```
void fl_set_oneliner_font(int style, int size);
```

See also Section 18.3 for similar but potentially multi-line message routines.

```
void fl_show_alert(const char *s1,const char *s2,const char *s3,int c)
```

Same as `fl_show_message()` except that an alert icon (!) is added. The extra parameter `c` controls whether to display the form centered on the screen.

```
int fl_show_question(const char *s1, const char *s2, const char *s3)
```

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(const char *s1,const char *s2,const char *s3,
            int numb,const char *b1,const char *b2,const char *b3)

void fl_set_choices_shortcut(const char *s1, const char *s2,
            const char *s3);
```

Shows a three line message with one, two or three buttons. `numb` indicates the number of buttons. `b1`, `b2` and `b3` 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> key to indicate the first, second, or third button. More mnemonic hotkeys can be defined using the shortcut routine, `s1,s2` and `s3` are the shortcuts to bind to the three buttons.

To change the font used in all messages, use the following routine

```
void fl_set_goodies_font(int style, int size)
```

To obtain some text from the user, use the following routine

```
const char *fl_show_input(const char *str1,const char *defstr)
```

This shows 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 `goodies.c`.c shows you these goodies.

It is possible to change some of the built-in button labels via the following the resource function with proper resource names

```
void fl_set_resource(const char *res_str, const char *value)
```

For example, to change the label of Dismiss button to “Go’" in the alert form, code similar to the following can be used after `fl_initialize` but before any use of the alert goodie:

```
fl_set_resource("flAlert.dismiss.label","Go");
```

Currently the following goodies resources are supported

```
flAlert.dismiss.label
flQuestion.yes.label
flQuestion.no.label
*.ok.label
```

6.2 Colormap

In a number of applications 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 decide 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 File selector

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

```
const char * fl_show_fselector(const char *message,const char *directory,
                               const char *pattern,const char *default)
```

A form will be shown in which listed are all files in directory `directory` that satisfy the pattern (See Fig 6.1.) `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. Function returns a pointer to a static buffer that contains the filename selected or null if the Cancel button is pressed (see below).

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. All directory entries read are cached internally (up to 10 directories), and if there is any change in directory entries, click on ReScan button to force an update.

Figure 6.1: File selector

There are total of `FL_MAX_FSELECTOR` (6) file selectors in the **Forms Library**with each having its own current directory and content cache. All the file selector functions documented manipulate the currently active file selector, which can be set using the following routine

```
int fl_use_fselector(int n)
```

where `n` is a number between 0 and `(FL_MAX_FSELECTOR - 1)`.

The window title of the file selector can be changed anytime using the following routine

```
void fl_set_fselector_title(const char *title)
```

To force an update programmatically, call

```
fl_invalidate_fselector_cache(void)
```

before `fl_show_fselector()`. Note that this call only forces an update once, and on the directory that is to be browsed. To disable caching altogether, the following routine can be used:

```
fl_disable_fselector_cache(int yes);
```

A false parameter (re)enables the directory cache.

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.

Complete keyboard navigation is built-in. E.g., you can use <ALT> `d` to change the directory instead of using the mouse.

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.

It is also possible to set a callback routine so that whenever the user double clicks on a filename, instead of returning the filename, the callback routine is invoked with the filename as the argument. To set such a callback, use the following routine

```
void fl_set_fselector_callback(void (*callback)(const char *, void *),
                                  void *user_data);
```

where the second argument of the `callback` is the `user_data`. Note that the behavior of the file selector is slightly different when a callback is present. Without the callback, a file selector is always modal.

The placement of the file selector is by default centered on the screen, which can be changed by the following routine

```
void fl_set_fselector_placement(int place);
```

where `place` is the placement request same as in `fl_show_form()`. The default is `FL_PLACE_CENTER|FL_FREE_SIZE`.

By default, an fselector is displayed with transient property set. To change the default, use the following routine

```
void fl_set_fselector_border(int flag)
```

set `border` request is the same as in `fl_show_form()`, but `FL_NOBORDER` is ignored.

When the arguments `directory`, `pattern` or `default` are empty, the previous value is used (some good defaults when this happens the first time). Thus the file selector "remembers" all the settings the selector used last time. The application program can figure out the directory, pattern and file name (without the path) after the user changed them using the routines

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

There are other routines that make the fselector more flexible. The most important of which is the ability to accommodate up to three application specific button:

```
void fl_add_fselector_appbutton(const char *label,
                                void (*callback)(void *), void *data)
```

Again, the argument `data` is passed to the `callback`

To remove an application specific button, use the following routine

```
void fl_remove_fselector_appbutton(const char *label)
```

Whenever this application specific button is pushed, the callback function is invoked. Within the callback function, in addition to using the routines mentioned above, the following routines can be used:

```
void fl_refresh_fselector(void)
```

This function causes the file selector to re-scan the current directory and to list all entries in it. For whatever reason, if there is a need to get the fselector form identifier, the following routine can be used:

```
FL_FORM *fl_get_fselector_form(void)
```

See `fbrowse.c` for the use of the file selector.

Although discouraged, it is recognized that direct access to the individual objects on the fselector form may be necessary. To this ends, the following routine exists

```
typedef struct
{
    FL_FORM *fselect;
    FL_OBJECT *browser, *input, *prompt, *resbutt;
    FL_OBJECT *patbutt, *dirbutt, *cancel, *ready;
} FD_FSELECTOR;

FD_FSELECTOR *fl_get_fselector_fdstruct(void)
```

You can, for example, change the default label strings of various buttons via structure members of `FD_FSELECTOR`:

```
FD_FSELECTOR *fs;
fs = fl_get_fselector_fdstruct();
fl_set_object_label(fs->ready,"Go !");
fl_fit_object_label(fs->ready, 2, 2);
```

Special files are marked with a distinct prefix in the browser (for example, `D` for directory, `p` for pipe etc). To change the prefix, use the following routine

```
void fl_set_fselector_filetype_marker(int dir, int fifo, int socket,
                                      int cdev, int bdev)
```

Although file systems under Unix are similar, they are not identical. In the implementation of fselector, the subtle differences in directory structure are isolated and conditionally compiled so an apparent uniform interface to the underlying directory structure is achieved. To facilitate alternative implementations of file selectors, the following (internal) routines can be freely used.

To get a directory listing, the following routine can be used

```
const FL_Dirlist *fl_get_dirlist(const char *dirname,
                                 const char *pattern,
                                 int *nfiles, int rescan)
```

where `dirname` is the directory name; `pattern` is a regular expression that is used to filter the directory entries; `nfiles` on return is the total number of entries in directory `dirname` that match the `pattern` (not exactly true, see below.) By default, directory entries are cached. A true `rescan` requests a re-read.

The `FL_Dirlist` is a structure defined as follows

```
typedef struct
{
    char *name;
    int type;
}  FL_Dirlist;
```

where `type` is one of the following file types

FT_FILE	a regular file.
FT_DIR	a directory.
FT_SOCK	a socket.
FT_FIFO	a pipe.
FT_LINK	a symbolic link.
FT_BLK	a block device.
FT_CHR	a character device.
FT_OTHER	?

To free the list cache returned by `fl_get_dirlist`, use the following call

```
void fl_free_dirlist(FL_Dirlist *dl)
```

Note that a cast may be required to rid of the `const` qualifier

By default, not all types of files are returned by `fl_get_dirlist`. The specific rules regarding which types of file to return are controlled by an additional filter after the pattern filter

```
int ffilter(const char *name, int type)
```

which is called for each entry (except for directory) that matches the pattern found in the directory. Function should return true if the entry is to be included in the dirlist. The default filter is similar to the following

```
int ffilter(const char *name, int type)
{
    return type == FT_FILE || type == FT_LINK;
}
```

To change the default filter, use the following routine

```
typedef int (*FL_DIRLIST_FILTER)(const char *, int);
FL_DIRLIST_FILTER fl_set_dirlist_filter(FL_DIRLIST_FILTER filter)
```

Since there is only one filter active at anytime in **XForms**, changing the filter affects file browser.

Part II

The Form Designer

Chapter 7

Introduction

This part of the documentation describes the **Form Designer**, a GUI builder meant to help you 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 I of this document.

Even though designing forms is quite easy and requires only a relatively small number of lines of C-code, it can be time consuming to figure out all required positions and sizes of the objects. The **Form Designer** was written to facilitate the construction of forms. With **Form Designer**, there is no longer any need to calculate or guess where the objects should be. The highly interactive and WYSIWYG (what you see is what you get) nature of the **Form Designer** relieves the application programmer from the time consuming process of user interface construction so that he/she can concentrate more on what the application program intends to accomplish.

Form Designer provides the abilities to interactively place, move and scale objects on a form, also the abilities to set all attributes of an object. Once satisfactory forms are constructed, 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_form_xxx()` for each form, where `xxx` indicates the form name. The application only needs to call it to generate the form designed. The code produced is easily readable.

The **Form Designer** also lets the user identify each object with C variables for later reference in the application program and allows advanced object callback bindings all within the **Form Designer**. 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.) A black window (the main window) will appear on the screen. This is the window in which you can create your forms. Next the control panel appears on the screen. No form is shown yet.

The control panel consists of five parts (see Fig. 8.1). The first part is the menu bar consisting of several groups of menus from which you can make selections or give commands to the program. 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 bottom 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 as we will come back to groups and their use later. 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. Each of these buttons is bound to a function key. 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 New Form on the top-left corner of the control panel just below the menu bar. 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 main 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 and drag the mouse 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. 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

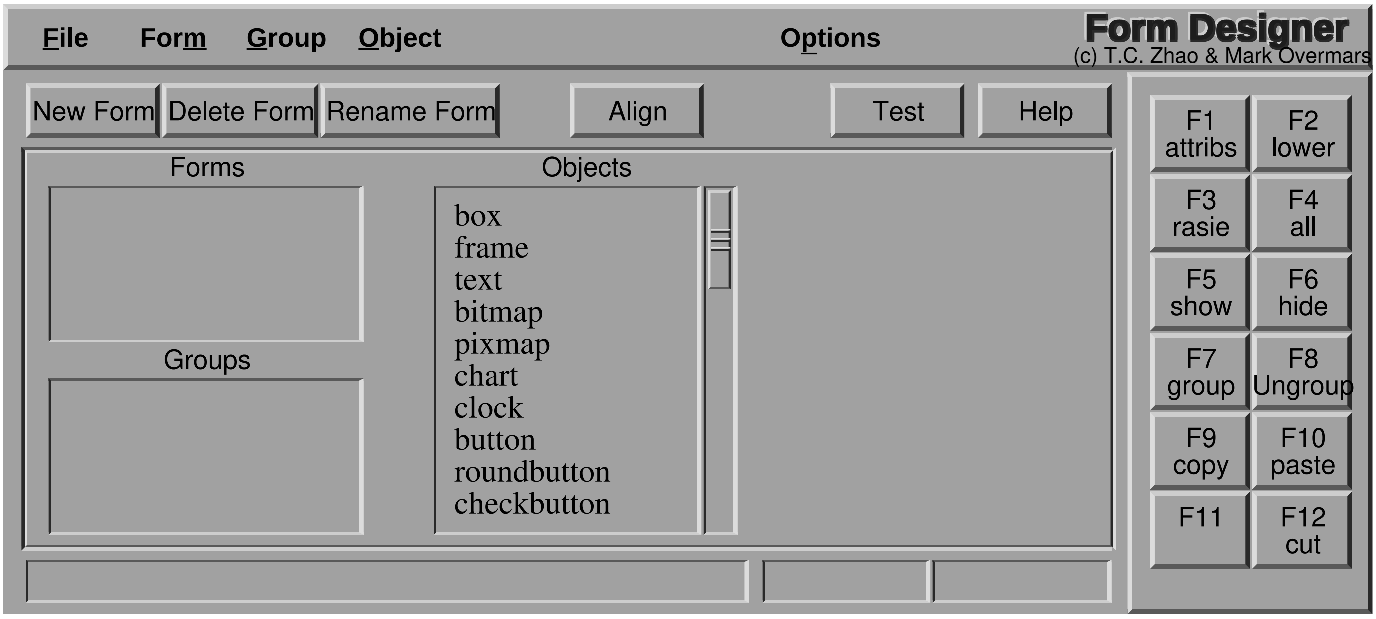

Figure 8.1: **Form Designer** control panel

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 and move or scale them simultaneously. See below for details.

To change the attributes, e.g., the label, of an object, press the right mouse button inside the object to select it. Next, press the function key <F1> (either on the keyboard or in the control form). This can also be achieved by double-clicking the right mouse button. A form will appear in which you can indicate all the different attributes. Their meanings should be clear (if you have read the documentation on the **Forms Library**). Change the attributes by pressing a 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 Accept. Or if you would like to preview how the new attributes look, you can press the Apply button. Press Restore to restore the original attributes. 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 choose *Save* from the File menu to save them to disk. 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 that the **Form Designer** can read it back in later. The application program now simply has to call the routine `create_form_xxx()` 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 [-xformoptions] [-fdesignoptions] [files[.fd]]
```

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

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

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

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

`-version` Prints current version and quits.

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

`-altformat` Generates an alternative output format.

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

`-unit` $point|pixel|mm|cp|cmm$ Outputs object sizes in units other than pixels. `cp` and `cmm` stand for centi-point (1/100 of a point) and centi-mm (1/100 of a milli-meter).

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

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

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

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

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

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

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

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

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

`fdesign` recognizes the following resources

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

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

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

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

Chapter 10

Creating Forms

10.1 Creating, changing and deleting forms

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

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

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

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

10.2 Adding objects

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

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

10.3 Selecting objects

To perform operations on objects that are already visible in the form, we first have to select them. 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 on the backface of the form. Although this is a normal object, it can not be treated in the same way as the other objects. It can be selected, but never in combination with other objects. Only two operations are allowed on it: changing its attributes and scaling it (which scales the size of the form).

10.4 Moving and scaling

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

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

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

10.5 Aligning objects

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

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.

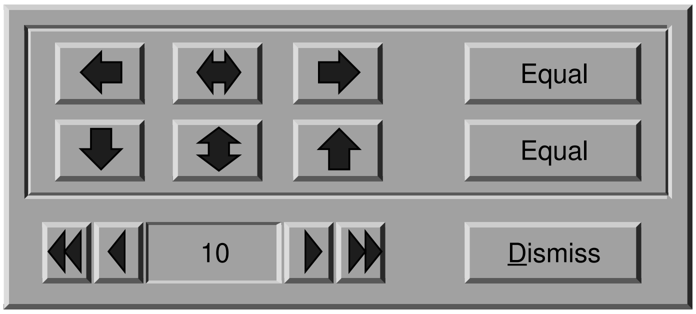

Figure 10.1: Object alignment control

In the alignment form you can also indicate the snapping size using the counter at the bottom. Choose 0 if you don't want to snap positions. Default the snapping is 10 pixels. Snapping helps in making objects the same size and in making 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 thus making the buttons invisible. This is definitely not what you want. The **Form Designer** makes it possible to raise objects (bring them to the top) or lower them (put them at the bottom). So you can lower the box to move it under the buttons. Raising or lowering objects is very simple. First select the objects using the right mouse button and next press the function key <F2> to lower the selection or <F3> to raise it.

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 the colors, 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, callback and argument. Name indicates the name of the object. If you type in a name here the object will be known to the application program under this name so that the program can refer to it. Take care that all object names used are different. They should be legal C variable names.[1] It is possible to use arrays of objects. E.g. if you define some objects as `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.)

Callback indicates the callback routine. If you type in something here, this routine will be bound to the object. In this case you also have to provide an argument that must be an integer (or cast to integer, as in `(long)&variable`). Of course, the application program will have to provide the callback routine.

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

10.9 Cut, Copy and Paste

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

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

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

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

press the function key <F7>. You will be prompted for the name of the group. This should be a legal C variable name (under which the group will be known to the application program) or should be empty. This name will be added to the list. In this way you can create many groups. Note that each object can be in only one group. So if you select it again and put it in a new group, it will be removed from its old group. Groups that become empty this way automatically disappear from the list. (When putting objects in a group they will be raised. This is unavoidable due to the structure of groups.)

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

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

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

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

10.11 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. It is better, though, to first group the objects before hiding them. Now you can select them by pressing the mouse on the group name in the group browser.

10.12 Testing forms

To test the current form, press the button labeled Test. The form will be displayed in the center of the screen. A panel will appear at the top right corner of the screen. This panel will show you the objects that will be returned and the callback routines called when working with the form. In this way you can verify whether the form behaves correctly and whether all objects have either callback routines or names (or both) associated with them. You can 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 select the item *Save* from the File menu. 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 and `ttt.h` contains all the object and form names as indicated by the user. It also contains declaration of the defined callback routines.

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

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

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

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

```
#ifndef FD_foobar_h_
#define FD_foobar_h_

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

typedef struct
{
     FL_FORM *foo;
```

```
    FL_OBJECT *f1obj1;
    FL_OBJECT *f1obj2;
    void *vdata;
    long ldata;
} FD_foo;

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

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

#endif /* FD_foobar_h */
```

and the corresponding C file:

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

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

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

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

    fdgui->bar = fl_bgn_form(....);
    fdgui->f2obj1 = fl_add_xxxx(....);
    .....
    fl_end_form();
    return fdui;
}
```

The application program would look something like the following:

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

/* add call back routines here */

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

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

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

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

As you see, `fdesign` generates a structure that groups together all objects on a particular form and the form itself. Application program can create more than one instances of the form if needed.

It is difficult to avoid globals in an event-driven callback scheme with the most difficulties occuring inside the callback function where another object on the same form may need to be accessed. Current setup makes it possible and relatively painless to achieve this. The easiest way of doing this is to use the `vdata` in `FD_` structure to hold the ID of the object needs to be accessed. In case of a need to access multiple objects, there is a field `u_vdata` in `FL_OBJECT` structure you can use. You simple use the field to hold the `FD_` structure:

```
fdfoo = create_form_foo();
fdfoo->f1obj1->u_vdata = fdfoo;
...
```

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

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

Not pretty, but adequate for practical purposes.

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

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

and use the callback as follows[1]

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

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

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

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

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

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

The alternative format outputs something like the following:

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

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

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

The C-routines:

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

[1]This scheme is probably illegal as pointers may be longer than a long although we do not know of any machines on which this is true.

```
FL_OBJECT *f2obj1, *f2obj2 ...;

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

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

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

Normally the application program would look something like this:

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

/* The call back routines */

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

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

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

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**.

All available object classes are summarized in Table 12.1.

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

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

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

For each object class there is also a routine

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

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

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

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

Name	Description
	Static Objects
Box	Rectangular areas to visually group objects.
Frame	A box with an empty inside region
Text	Simple one line labels.
Bitmap	Displays an X11 bitmap.
Pixmap	Displays a pixmap using the XPM library.
Clock	A clock.
Chart	Bar-charts, pie-charts, strip-charts, etc.
	Button Like Objects
Button	Many different kinds and types of buttons that the user can push to indicate certain settings or actions.
	Valuator Objects
Slider	Both vertical and horizontal sliders to let the user indicate some float value.
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.
XYPlot	XYPlot shows simple 2D xy-plot from a tabulated function or a datafile. Data points can be interactively manipulated and retrieved.
Pop-ups	Pop-ups are used by menu and choice. it can also be used stand-alone to allow the user to make selections among many choices.
Canvas	Canvases are managed plain X windows. It differs from a raw application window only in the way its geometry is managed, not in the way various interaction is set up.

Table 12.1: List of object classes

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

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

Chapter 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, FL_Coord x, FL_Coord y,
                      FL_Coord w, FL_Coord h, const char *label)
```

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

Types

The following types are available:

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

Interaction

No interaction takes place with boxes.

Other routines

No other routines are available for boxes.

Attributes

Color1 controls the color of the box.

Remarks

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

13.2 Frame

Short description

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

Adding an object

To add an frame to a form you use the routine

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

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

Types

The following types are available:

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

Interaction

No interaction takes place with frames.

Other routines

None.

Attributes

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

Remarks

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

13.3 Text

Short description

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

Adding an object

To add a text to a form you use the routine

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

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

Types

Only one type of text exists: FL_NORMAL_TEXT.

Interaction

No interaction takes place with text objects.

Other routines

No other routines are available for texts.

Attributes

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

Remarks

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

13.4 Bitmap

Short description

A bitmap is a simple bitmap shown on a form.

Adding an object

To add a bitmap to a form you use the routine

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

The meaning of the parameters is as usual. The label is by default placed below the bitmap. The bitmap will be empty.

Types

Only the type FL_NORMAL_BITMAP is available.

Interaction

No interaction takes place with a bitmap. For bitmap that interacts, see Section 14.1 on bitmapbutton. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static bitmap.)

Other routines

To set the actual bitmap being displayed use

```
void fl_set_bitmap_data(FL_OBJECT *ob,int width,int height,char *bits)
void fl_set_bitmap_file(FL_OBJECT *ob,const char *file);
```

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

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

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

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

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

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

Attributes

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

Remarks

See demo33.c for a demo of a bitmap.

13.5 Pixmap

Short description

A pixmap is a simple pixmap shown on a form.

Adding an object

To add a bitmap to a form you use the routine

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

The meaning of the parameters is as usual. The label is by default placed below the pixmap. The pixmap will be empty.

Types

Only the type `FL_NORMAL_PIXMAP` is available.

Interaction

No interaction takes place with a pixmap. For pixmap that interacts, see Section 14.1 on `pixmapbutton`. (You can also place a hidden button over it if you want something to happen when pressing the mouse on a static pixmap.)

Other routines

To set the actual pixmap being displayed use

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

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

Where `data` contains the pixmap data as an array of char pointers and `file` is the name of the file that contains the pixmap data. Note that both of these functions do *not* free the old pixmaps associated with the object. If you're writing a pixmap browser type applications, be sure to free the old pixmaps using `fl_free_pixmap_pixmap` prior to calling these two routines.

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

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

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

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

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

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

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

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

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

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

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

Attributes

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

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

Remarks

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

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

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

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

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

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

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

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

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

You will need the XPM library developed by Arnaud Le Hors and GROUPE BULL (`lehors@sophia.inria.fr`) to use pixmap. XPM library can be obtained from many X distribution/mirror sites via anonymous ftp (ftp://ftp.x.org/contrib and ftp://avahi.inria.fr/pub/xpm are the official sites).

13.6 Clock

Short description

A clock object simply displays a clock on the form.

Adding an object

To add a clock to a form you use the routine

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

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

Types

The following types are available:

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

Interaction

No interaction takes place with clocks.

Other routines

To get the 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 digital clock.

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

Remarks

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

13.7 Chart

Short description

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

Adding an object

To add a chart object to a form use the routine

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

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

Types

The following types are available:

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

All charts except pie-charts can display positive and negative data. Pie-charts will ignore values that are ≤ 0. The maximal number of values displayed in the chart can be set using the routine `fl_set_chart_maxnumb()`. The number must be bounded by `FL_CHART_MAX` which is 256. Switching between different types can be done without any complications.

Interaction

No interaction takes place with charts.

Other routines

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

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

To add an item to a chart use

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

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

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

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

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

To replace the value of a particular item use the routine

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

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

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

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

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

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

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

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

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

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

Attributes

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

Remarks

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

Chapter 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.

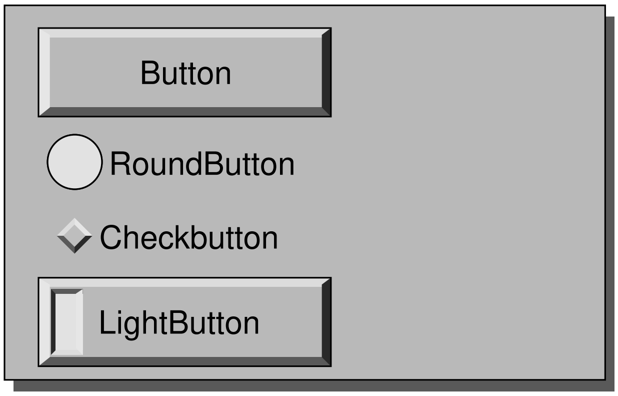

Figure 14.1: Button Classes

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

Adding an object

To add buttons use one of the following routines:

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

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

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

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

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

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

The meaning of the parameters is as usual. The label is by default placed inside the button for button and lightbutton. For roundbutton, bitmapbutton and pixmapbutton, it is placed to the right of the circle and to the bottom of the bitmap/pixmap respectively.

Types

The following types of buttons are available:

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

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

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

Interaction

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

Other routines

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

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

pushed indicates whether the button should be pushed (1) or released (0). 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. Further, this routine only simulates the visual appearance and perhaps some internal states, it does *not* affect the program flow in anyway, i.e., setting a button being pushed does not invoke its callback or results in the button returned to the program. For that, fl_trigger_object() is needed.

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 (or release) use the routine

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

It returns the number of the mouse button (1 = left, 2 = middle, 3 = right). If the last push is triggered by a shortcut (see below), the function returns the keysym (ascii value if ASCII) of the key plus FL_SHORTCUT. For example, if a button has a shortcut <CNTRL> C, the button number returned upon activation of the shortcut would be FL_SHORTCUT 3+.

If more information is desired about the last event, use

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

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

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

Note that str is a string, not a character. This string should contain all the characters that correspond to this button. So, e.g., if you use string "^QQq" the button will react on the keys q, Q and <CNTRL> Q. (As you see you should use the symbol ^ to indicate the control key. Similarly you can use the symbol # to indicate the <ALT> key.) Be careful with your choices. When the form also contains input fields you probably don't want to use the normal printable characters because they can no longer be used for input in the input fields. Shortcuts always go before input fields. Other special keys, such as <F1> etc., can also be used as shortcuts. See Section 21.1 for details. Finally realize that a return button is in fact a normal button with the <Return> key as a shortcut. So don't change the shortcuts for such a button.

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

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

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

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

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

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

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

Note that these routines do not free the pixmaps already associated with the button. To free the pixmaps, use the following routine

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

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

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

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

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

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

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

See also Section 13.5 for pixmap color and transparency handling.

Attributes

For normal buttons color1 controls the normal color and color2 the color when pushed. For lightbuttons color1 is the color of the light when off and color2 the color when on. For round buttons, color1 is the color of the circle and color2 the color of the circle that is placed inside it when pushed. For bitmapbuttons, color1 is the normal box color (or bitmap background if nobox) and color2 is used to indicate the focus color. The foreground color of the bitmap is controlled by label color.

Remarks

See all demo programs, in particular `pushbutton.c` and `buttonall.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, FL_Coord x, FL_Coord y,
                         FL_Coord w, FL_Coord h, const char *label)
```

or

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

The meaning of the parameters is as usual. The label is by default placed below the slider. The second type of slider displays its value above or to the left of the slider.

Types

The following types of sliders are available:

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

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

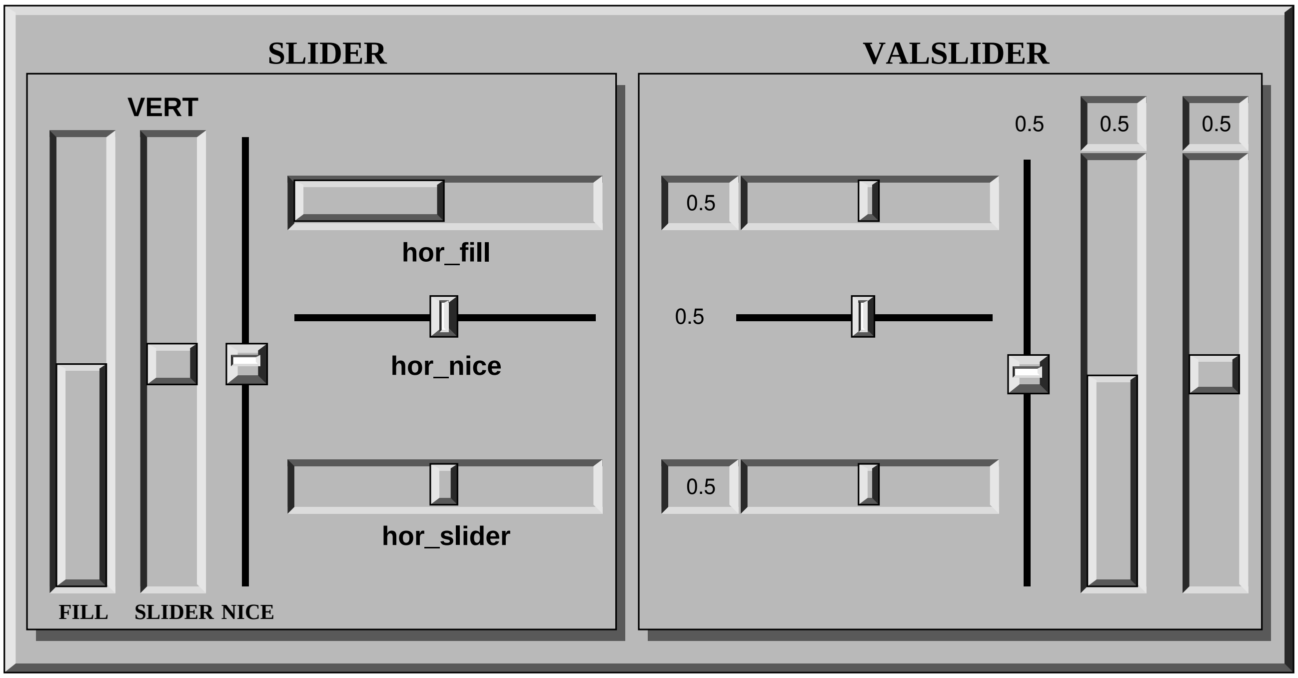

Figure 15.1: All sliders

Interaction

Whenever the user changes the value of the slider using the mouse, the slider is returned (or the callback called) by the interaction routines. The slider position is changed by moving the mouse inside the slider area. For fine control, hold down the left or right <SHIFT> key while moving the slider.

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

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

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

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

`FL_RETURN_CHANGED` return whenever the slider value is changed.

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

FL_RETURN_ALWAYS return all the time. Not very useful.

Other routines

To change the value of the slider use the routines

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

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

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

To obtain the current values of the slider use

```
double fl_get_slider_value(FL_OBJECT *obj)

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

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

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

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

Attributes

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

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

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

`size` should be a float between 0.0 and 1.0. With size=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`.

The routine

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

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

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

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

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

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

Remarks

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

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

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

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

Types

The following types of dials are available:

`FL_NORMAL_DIAL`	A dial with a knob indicating the position.
`FL_LINE_DIAL`	A dial with a line indicating the position.

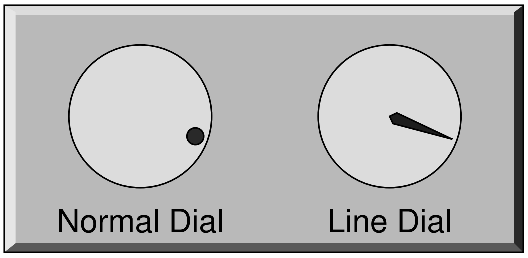

Figure 15.2: Types of dials

Interaction

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

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, double val)

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

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

```
double fl_get_dial_value(FL_OBJECT *obj)

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

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

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

where `thetai` maps to the minimum value of the dial and `thetaf` maps to the maximum value of the dial. By default, the minimum angle is 0 and the maximum angle is 360.

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

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

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

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

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

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

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

See the demo program `ldial.c` and `ndial.c` for examples 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-values.

Adding an object

A positioner can be added to a form using the call

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

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

Types

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 0 to achieve this goal.

Other routines

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

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

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

void fl_set_positioner_yvalue(FL_OBJECT *obj, double val)

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

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

To obtain the current values of the positioner use

```
double fl_get_positioner_xvalue(FL_OBJECT *obj)

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

```
double fl_get_positioner_yvalue(FL_OBJECT *obj)

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

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

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

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

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

Attributes

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

Remarks

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

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

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

Types

The following types of counters are available:

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

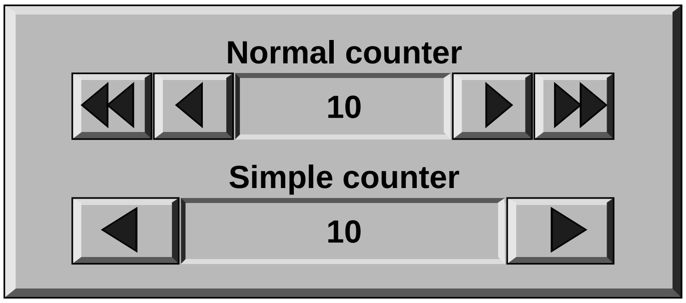

Figure 15.3: Counter types

Interaction

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

In some applications you might want the counter to be returned to the application program whenever the value changes. To achieve this call the routine

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

Set `always` to TRUE to achieve this.

Other routines

To change the value of the counter use the routines

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

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

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

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

To obtain the current value of the counter use

```
double 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

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

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

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

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

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

Attributes

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

Remarks

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

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

Chapter 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, FL_Coord x, FL_Coord y,
                        FL_Coord w, FL_Coord h, const char *label)
```

The meaning of the parameters is as usual. The label is by default placed in front of the input field.

Types

The following types of input fields exist:

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

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

Note that multi-line input fields do not scroll (neither horizontally nor vertically). So the area reserved for the field should be large enough to contain the entire text.

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 following keys (as in *emacs*(1)) for maneuvering inside the input field:

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

In addition to the above keys, <CNTRL> `L` can be used to refresh the input field.

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

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

When the user presses the <TAB> key the input field is returned to the application program and the input focus is directed to the next input field. This also happens when the user presses the <RETURN> key but only if the form does not contain a return button. This does not work for multi-line input fields where return key is used to separate lines. Also when the user picks a new input field with the mouse, the current input object is returned.

The above mechanism is the default behavior of an input field. Depending on the application, other options might be useful. To change the precise condition for the object to be returned (or equivalently the callback invoked), the following function can be used:[1].

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

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

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

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

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

`FL_RETURN_ALWAYS` Invoke the callback function upon each keystroke.
modified or not.

Although an input with `FL_RETURN_ALWAYS` attributes can be used to limit the type of characters that can be entered into the input field, use of the following is typically more appropriate

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

The filter function is called whenever a new (regular) character is entered. `old` is the string in the input field before the newly typed character `c` is merged into `cur`. If the new character is not an acceptable character for the input field, the `filter` function should return `FL_INVALID` otherwise `FL_VALID`. If `FL_INVALID` is returned, the new character is discarded and the input field remains unmodified. The function returns the old filter. Unlike the built-in filters, keyboard bell is not sound when `FL_INVALID` is received. To sound the bell, return `FL_INVALID|FL_RINGBELL`.

Finally, there is a routine that can be used to limit the number of characters per line

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

To reset the limit to infinite, set `maxchars` to 0. Currently this function only works in `FL_NORMAL_INPUT`.

[1] This routine is obsolescent. Use `fl_set_object_return` instead

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[])
```

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

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

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

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

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

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

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

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

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

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

where `xpos` and `ypos` are measured in characters (lines). E.g., given the input field, `"an arbitrary string"`, the call `fl_set_input_cursorpos(ob, 4, y)` places the the cursor after the character <A> in `arbitrary`. Note currently this function only works correctly in `FL_NORMAL_INPUT` and does not do anything with `ypos`.

There is also a function that returns the cursor position within the input field

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

Where `xpos` and `ypos` are the cursor positions measured in characters (lines) and `xpos` always gives the number of characters in front of the cursor. If the input field does not input focus (thus has no cursor), the `xpos` is set to -1. Currently this function works for `FL_NORMAL_INPUT` and `ypos` always returns 1.

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

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

Scrolling might not always be desirable and you can disable input field scrolling using the following routine with a false `flag`

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

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 `minput.c` for multi-line input fields. See `secretinput.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) a pop-up menu appears. The user can then make a selection from the menu.

Adding an object

To add a menu to a form use the routine

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

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

Types

The following types are available:

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

PUSH_MENU and PULLDOWN_MENU behaves in exactly the same way. The only difference is in the way they are drawn when the menu is active: PUSH_MENU's menu appears to be an up_box casting a shadow while PULLDOWN_MENU's is just an extension of the menu box (see Fig. 17.1).

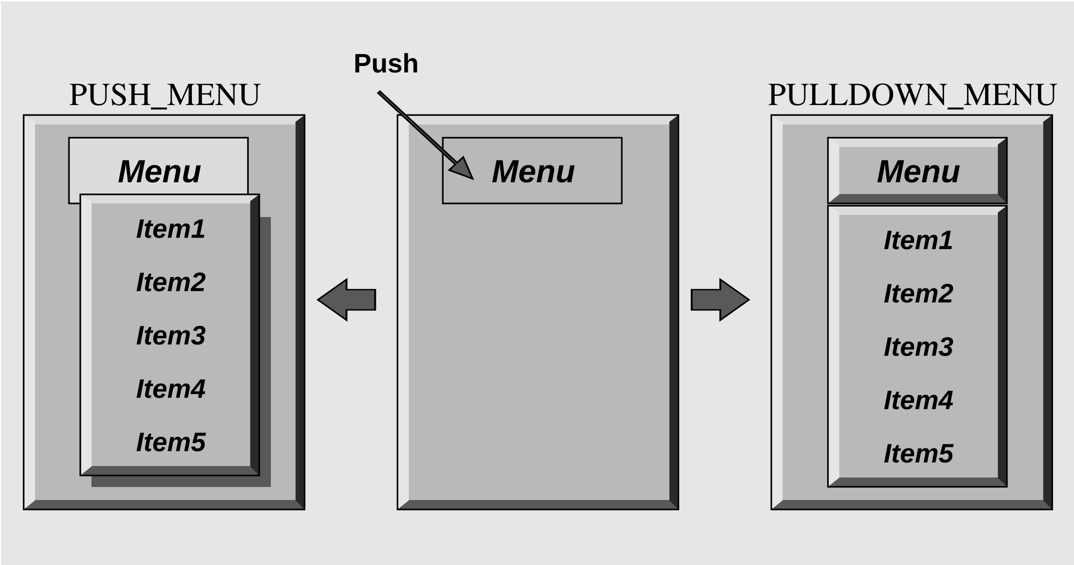

Figure 17.1: Menu types

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, const char *menustr)
```

`menustr` describes the menu in the form used by XPopups (See Section 18.3. In short, it should contain the menu items, separated by a bar, e.g., "First|Second|Third". See Section 18.3 for special tags that can be used to indicate special attributes (radio, toggle for example). Whenever the user selects some menu item, the menu object is returned to the application program. To find the actual menu item selected by the user use

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

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

You can also obtain the text of the item selected

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

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

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

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

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

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

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

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

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

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

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

to clear the whole menu use the routine:

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

One can change the appearance of different menu items. In particular, it is 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, unsigned mode)
```

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

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

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

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

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

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

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

Finally there is the routine:

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

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

Attributes

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

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

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

```
void fl_setpup_fontsize(int size)

void fl_setpup_fontstyle(int style)
```

Currently there is not direct support for cascade menus in menu class. You can, however, get a cascade menu by using a popup and then attach the popup ID to a menu object via the following routine

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

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

Remarks

See `menu.c` for an example of the use of menus. Cascade menus are not directly supported. You can use `FL_MENU_BUTTON` to initiate a callback and use XPopup directly within the callback. See `pup.c` for an example of this approach.

17.2 Choice

Short description

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

Adding an object

To add a choice object to a form use the routine

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

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

Types

The following types are available:

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

Interaction

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

Other routines

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

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

To add a line to a choice object use

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

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

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

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

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

NULL is returned when there is no current choice.

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

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

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

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

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

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

Here *mode* is the same as that used in menu object. See also Section 18.3 for details.

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

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

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

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

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

You can select `item2` by using

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

or

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

Note the spaces in the text.

Attributes

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

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

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

To set the font size used inside the choice object use

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

To set the font style used inside the choice object use

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

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

```
void fl_setpup_fontsize(int size)

void fl_setpup_fontstyle(int style)
```

See section 3.11.3 for details on font sizes and styles.

Remarks

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

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 the text does not fit inside the box, a scroll bar is automatically added so that the user can scroll through it. A browser can be used for building up a help facility or to give messages to the user.

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

Adding an object

To add a browser to a form use the routine

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

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

Types

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

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

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

Interaction

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

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

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

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

Other routines

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

To make a browser empty use:

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

To add a line to a browser use

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

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

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

The difference is that with this call the browser will be shifted such that the newly 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,
                            const 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,
                             const char *text)
```

Making many changes to a visible browser at the same moment, i.e., clearing it and loading it with a number of new choices, is 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 the following

```
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

```
const 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, const char *filename)
```

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

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

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

void fl_deselect_browser_line(FL_OBJECT *obj, int line)

void fl_deselect_browser(FL_OBJECT *obj)
```

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

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

The routine

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

where `xoff` is in pixels.

Attributes

Never use the boxtype `FL_NO_BOX` for browsers.

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

To set the font size used inside the browser use

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

To set the font style used inside the browser use

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

See section 3.11.3 for details on font sizes and styles.

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

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

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

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

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

To align different text fields on a line, tabs (`\t`) can be embedded in the text.

By default, the scroll bar is drawn to the right of the browser. It can easily be changed to be drawn to the left of the browser:

```
void fl_set_browser_leftslider(FL_OBJECT *ob, int yes)
```

There is also a routine that can be used to turn the (vertical) scrollbar on and off

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

The following table gives the possible values and their meanings:

FL_ON	turns the scrollbar on if there are enough lines in the browser.
FL_OFF	turns the scrollbar off.
FL_ALWAYS_ON	turns the scrollbar on.

Note `FL_ALWAYS_ON` is not currently supported.

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

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

where `x` and `y` are measured from the top-left corner of the form (or the smallest enclosing window).

Remarks

There is currently a limit of maximum 1024 bytes per line for `fl_load_browser()`.

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

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

The meaning of the parameters is as usual.

Types

There are at the moment three types of timers:

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

Interaction

When a visible timer expires it starts blinking. The user can stop the blinking by pressing the mouse on it. 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, double 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

```
double 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 `timer.c` for the use of timers.

18.2 XYPlot

Short description

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

Adding an object

To add an xyplot object to a form use the routine

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

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

Types

The following types are available:

FL_NORMAL_XYPLOT	solid line
FL_SQUARE_XYPLOT	solid line plus squares on data points
FL_CIRCLE_XYPLOT	solid line plus circles on data points
FL_FILLED_XYPLOT	the area under the curve is filled
FL_POINTS_XYPLOT	only data points are shown
FL_DASHED_XYPLOT	dashed line
FL_IMPULSE_XYPLOT	vertical line
FL_ACTIVE_XYPLOT	accepts manipulations
FL_EMPTY_XYPLOT	draws only the axes

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

Interaction

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

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

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

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

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

To set or replace the data for an xyplot, use

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

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

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

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

The data file should be an ASCII file consisting of data lines. Each data line must have two columns indicating the (x,y) pair with space, tab or comma (,) separating the two columns. Lines that start with `!` `;` `#` are considered to be comments and are ignored.

To get a copy of the current `FL_XYPLOT` data, use

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

The caller must supply the space for the data.

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

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

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

Other routines

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

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

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

Note that the actual scaling routine may choose a value other than that requested if it decides that this would make the plot look nicer, thus `major minor` can only be taken as a hint to the scaling routine. However, in no case will the scaling routine generate a major that differs from the requested value by more than 3.

To change the size of the symbols draw on the data point, use the following routine

```
void fl_set_xyplot_symbolsize(FL_OBJECT *ob, int size)
```

where `size` should be given in pixels. The default is 4.

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

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

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

Data that fall outside of the range will be clipped. To restore autoscaling, use `max<=min`.

To get the current bounds, use the following routines

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

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

To replace the value of a particular point use the routine

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

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

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

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

where `x` and `y` are the coordinates where text is to be placed and `align` specifies the placement options relative to the specified point. (See `fl_set_object_lalign()` for valid options).

To remove an inset text, use the following routine

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

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

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

where `ID` must be between 1 and `FL_MAX_XYPLOTOVERLAY` (32) inclusive. Again, the data is copied internally (old data freed if any) The type (`FL_NORMAL_XYPLOT` etc.) used is the same as the object itself. To change an overlay style, use the following call

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

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

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

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

The function returns the previous maximum number of overlays.

To delete an overlay, use the following routine

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

Data may be interpolated using nth order Lagrangian polynomial

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

where `ID` is the overlay ID (use 0 for the original data) of the xyplot; `degree` is the order of the polynomial to use and `grid` is the working grid onto which the data are to be interpolated.

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

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

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

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

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

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

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

$$screenCoord = a \times data + b \qquad (linearscale)$$
$$screenCoord = a \times \log_p(data) + b \qquad (logscale)$$

where p is the base[1].

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

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

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

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

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

Finally there are routines to change the font size and style for labels, inset text etc:

```
fl_set_xyplot_fontsize(FL_OBJECT *ob, int size);

fl_set_xyplot_fontstyle(FL_OBJECT *ob, int style);
```

Remarks

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

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

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

If successful, the function returns the number of points ((inx[num_in-1] - inx[0]) / grid + 1.01) in the interpolated function else it returns -1.

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

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

18.3 Pop-ups

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

Short description

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

Define a new popup

To define a new popup, use the following routines

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

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

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

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

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

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

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

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

`%i` Makes this item inactive.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

To display a popup, use the following routine

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

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

A typical procedure may look as follows:

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

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

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

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

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

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

Interaction

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

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

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

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

where `item_val` is the value `fl_dopup()` would have returned if that item was selected (%x or position) and `hotkeys` is a string specifying all the hotkey combinations. See Section 21.1 for details. Briefly, `#` and `^` denote respectively the <ALT> and the <CONTROL> key. `&n` where n=1,2, etc., can be used to denote the function key `n`. Thus if `hotkeys` is set to `"#a^A`, both <CNTRL> `A` and <ALT> `A` are bound to the item. One additional property of the hotkey is the underlining of corresponding letters in the item string. Again, only the first alphabet in the hotkey is used. Therefore, for item string `"A Choice"`, hotkey `"Cc"`, `"#C"` or `"^C"` will result in the `C` in `"A Choice"` being underlined while `"cC"` and `"#c"` will not.

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

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

Similar function exists to associate a menu item with a submenu

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

Other routines

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

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

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

The following modes (and bitwise ORing thereof) are available

FL_PUP_NONE No special characteristics. The default.

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

FL_PUP_BOX Entry has an empty box to the left.

FL_PUP_CHECK Entry has a box to the left.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Attributes

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

```
void fl_setpup_fontsize(int size);
void fl_setpup_fontstyle(int style);
```

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

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

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

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

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

For example, after the following sequence,

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

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

The appearance of popups can be change by the following routines

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

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

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

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

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

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

```
void fl_setpup_checkcolor(FL_COLOR checkcolor)
```

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

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

The function returns the current limit.

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

```
void fl_showpup(int menuID)
void fl_hidepup(int menuID)
```

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

Remarks

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

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

18.4 Canvas

Scrolled canvas is not functional yet.

Short description

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

Adding an object

To add a canvas to a form you use the routine

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

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

Types

The following types of canvases exist:

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

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

Interaction

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

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

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

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

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

The `fl_add_canvas_handler()` function registers a procedure with the event dispatching system of the **Forms Library**. The library takes care of soliciting the proper event from the server. Other book keeping (e.g., drawing the box that encloses the canvas, etc.) is done by the object handler.

To remove a registered handler, use the following routine

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

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

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

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

or use the macro

```
FL_ObjWin(canvas_object)
```

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

Other routines

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

```
void fl_set_canvas_attributes(FL_OBJECT *ob, Visual *visual,
                              int depth, unsigned mask,
                              XSetWindowAttributes *xswa);
```

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

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

Colormap fl_get_canvas_colormap(FL_OBJECT *ob)

void fl_set_canvas_visual(FL_OBJECT *ob, Visual *vi)

void fl_set_canvas_depth(FL_OBJECT *ob, int depth)

int fl_get_canvas_depth(FL_OBJECT *ob)
```

By default, canvases are decorated with an `FL_DOWN_FRAME`. To change the decoration, use the following routine

```
void fl_set_canvas_decoration(FL_OBJECT *ob, int frame_type)
```

See Section 13.2 for a list of available frame types. Note not all frame types are appropriate for decorations.

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

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

where `n_colors` indicates how many colors in the newly created colormap should be filled with **XForms**'s default colors (to avoid flashing).

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

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

Attributes

None of the attributes, such as boxtype and object color, apply to the canvas class.

OpenGL canvases

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

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

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

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

A glcanvas so created will have the following attributes by default

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

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

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

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

To get the current defaults, use the following routine

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

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

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

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

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

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

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

The caller must supply the space for the attribute values.

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

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

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

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

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

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

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

Remarks

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

All specialized canvases are created by creating a generic canvas first

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

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

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

where `init` will be called before the creation of the canvas window; `activate` is called once the window is created; `cleanup` is called before the window is destroyed.

This routine can also be used to modify a canvas after it is added to a form but before the form is shown:

```
canvas = fl_add_canvas(type, x, y, w, h, "canvas");
fl_modify_canvas_prop(canvas, myInit, myActivate, myCleanup);
```

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

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

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

int glx_init(FL_OBJECT *ob)
{
    XVisual *vi;
    GLXContext context;

    /* query for OpenGL capabilities */

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

    /* select the desired visual */

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

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

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

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

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

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

```
int glx_activate(FL_OBJECT * ob)
{
    glXMakeCurrent(fl_display, FL_ObjWin(ob), ob->c_vdata);
    return 0;
}

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

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

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

```
}

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

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

Part IV

Designing your own object classes

Chapter 19

Introduction

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

Creating such new object classes and adding them to the library is simpler than it sounds. In fact it is almost the same as making free objects. This part gives you all the details of how to add new classes. In chapter 20 a global architectural overview is given of how the **Forms Library** works and how it communicates with the different object classes by means of events (messages). Chapter 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, a role equivalent to a superclass (thus all other object classes have `FL_OBJECT` as their parent class) in object-oriented programming.

One of the important aspects of an object is how to draw it on the screen. Chapter 23 gives all the details on drawing objects. The **Forms Library** contains a large number of routines that help you draw objects. In this chapter an overview is given of all of them. Chapter 24 gives an example illustrating how to create a new object class. Due to the importance of button classes, special routines are provided by the **Forms Library** to facilitate the creation of this particular class of objects. Chapter 25 illustrates by two examples the procedures of creating new button classes using the special services. One of the examples is taken from the **Forms Library** itself and the other offers actual usability.

Sometimes it might be desirable to alter the behavior of a built-in class slightly. Obviously a full-blown (re)implementation from scratch of the original object class is not warranted. Chapter 26.1 discusses the possibilities of using the pre-emptive handler of an object to implement derived objects.

Chapter 20

Global structure

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

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

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

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

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

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

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

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

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

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

This routine is the same as the handle routine for free objects and should handle particular events for the object. `mx`, `my` give the current mouse position and *key* the key that was pressed (if this information is related to the event). See chapter 21 for the types of events and the actions that should be taken. `xev` is the XEvent that caused the invocation of the handler. Note that some of the events may have a null `xev` parameter, so `xev` should be checked before dereferencing it.

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

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

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

that sets particular values for the object and

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

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

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 and also more flexible, there should in fact be two routines: `fl_create_NEW()` that creates the object and `fl_add_NEW()` that actually adds it to the form. They normally look as follows:

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

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

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

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

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

The constant `FL_NEW` will indicate the object class. It should be an integer. The numbers 0–1000 and 10000 and higher are reserved for the system and should not be used. Also it is preferable to use `fl_malloc()`, `fl_calloc()`, `fl_realloc()` and `fl_free()` to allocate/free the memory for the instance specific structures as future versions of **Forms Library** might offer debugging/safer version of these functions.

The pointer `ob` returned by `fl_make_object()` will have all of its fields set to some defaults (See Chapter 22). In other words, the newly created object inherits many attributes of a generic one. Any class specific defaults that are different from the generic one can be changed after `fl_make_object()`. Conversion of unit, if different from the default `pixel`, is performed within `fl_make_object()` and a class module never needs to know what the prevailing unit is. After the object is created, it has to be added to a form:

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

Chapter 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 for which action must be taken. In the following we assume that `obj` is the object to which the event is sent. The following types of events can be sent to an object:

`FL_DRAW` The object has to be redrawn. To figure out the size of the object you can use the fields `obj->x`, `obj->y`, `obj->w` and `obj->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. This can be figured out as follows. The field `obj->belowmouse` indicates whether the object is below the mouse. The field `obj->pushed` indicates whether the object is currently being pushed with the mouse. Finally, `obj->focus` indicate whether input focus is directed towards this object. Note that the drawing of the object is the full responsibility of the object class. So e.g. also the bounding box and the label, which can be found in the field `obj->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_DRAWLABEL` This event typically follows `FL_DRAW` and indicates the object label needs to be (re)drawn. If the object in question always draws its label inside the bounding box, and is taken care of by `FL_DRAW`, you can ignore this event.

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

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

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

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

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

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

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

FL_MOUSE The mouse position has changed (while a button is down). This event is sent to an object between an FL_PUSH and an FL_RELEASE event. The mouse position is given with the routine and action can be taken. For example, sliders use this event while 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->input is set to 1 (see below).

FL_UNFOCUS Input is no longer focussed on this object.

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

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

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

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

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

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

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. Obviously only active objects can have shortcuts. At the moment there are three object classes that use this, namely buttons, inputs and browsers although they behave differently.

The mechanism works as follows. There is a routine

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

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

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

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

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

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

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

Now whenever the user presses one of these keys an `FL_SHORTCUT` event is sent to the object. Here the key pressed is given with the handle routine (in the argument `key`). Combinations with the <ALT> key are given by adding `FL_ALT_VAL` (currently the 17th bit, i.e., 0x10000) to the ASCII value of the rest. E.g. `#^E` is passed as 5+`FL_ALT_VAL`. The object can now take action accordingly. If you use shortcuts to manipulate class object specific things, you will need to create a routine to communicate with the user, e.g., `fl_set_NEW_shortcut()`, and do your own internal bookkeeping to track what keys do what and then call `fl_set_object_shortcut()` to register the shortcut in the event dispatching module. The idea is NOT that the user himself calls `fl_set_object_shortcut()` but that the class provides a routine for this that also keeps track of the required internal bookkeeping. Of course, if there is no internal bookkeeping, a macro to this effect would suffice. For example, `fl_set_button_shortcut` is defined as `fl_set_object_shortcut`.

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

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

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

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

FL_KEY_SPECIAL All special keys (> 255).

FL_KEY_ALL All keys.

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

Chapter 22

The type FL_OBJECT

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`wantkey` An input object normally does not receive <TAB> or <RETURN> keystrokes or any other keys except those that have values between 0-255 and left- and right-arrows

(<TAB> and <RETURN> are reserved and used to switch between input objects). By setting this field to `FL_KEY_TAB` these keystrokes as well as as four directional cursor keys will also be sent to the object when focus is directed to it. If however, an object is only interested in keys that are special (e.g., <HOME>,<PGUP> etc), this variable can be set to `FL_KEY_SPECIAL` with or without `input` being set.

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

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

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

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

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

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

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

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

`c_vdata` A void pointer for the class routine. The main module does not reference or modify this field in any way.

`c_ldata` A long variable for the class routine. The main module does not reference or modify this field in any way.

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

`u_ldata` A long variable provided for the application program.

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

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

The generic object construction routine

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

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

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

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

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

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

`int deactivated` indicates if the form is deactivated.

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

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

`Window window` The form window.

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

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

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

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

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

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

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

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

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

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

```
int fl_get_form_vclass(FL_FORM *);

int fl_get_vclass(void);
```

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

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

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

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

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

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

There are other routines that might be useful:

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

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

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

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

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

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

Or more conveniently

```
fl_color(FL_RED);
```

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

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

```
fl_bk_color(FL_COLOR index)
```

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

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

which is equivalent to the following Xlib calls

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
void fl_unset_clipping(void)
```

to stop clipping.

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

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

void fl_unset_text_clipping(void)
```

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

```
void fl_winset(Window win)
Window fl_winget(void)
```

The most basic drawing routine is the rectangle routines:

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

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

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

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

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

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

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

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

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

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

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

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

`fl_polyf()` draws a filled polygon; `fl_polyl()` draws a polyline; and `fl_polybound()` draws a filled polygon with a black outline. *Note all polygon routines require that* `xpoint` *have spaces to hold* `n+1` *points*.

To draw an ellipse, either filled or open, the following routines can be used (use `w == h` to get a circle)

```
void fl_ovalf(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)

void fl_ovall(FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,FL_COLOR c)

void fl_ovalbound(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                  FL_COLOR c)
```

To draw circular arcs, either open or filled, the following routines can be used

```
void fl_arc(FL_Coord x, FL_Coord y, FL_Coord radius,
            int start_theta, int end_theta, FL_COLOR col)

void fl_arcf(FL_Coord x, FL_Coord y, FL_Coord radius,
            int thetai, int thetaf, FL_COLOR col)
```

where thetai and thetaf are the starting and ending angles of the arc, in unit of one tenth of a degree (1/10 degree); and x,y are the center of the arc. If $\theta_f - \theta_i$ is larger than 3600 (360 degrees), it is truncated to 360 degrees.

To draw elliptical arcs, the following routine should be used

```
void fl_pieslice(int fill, FL_Coord x, FL_Coord y, FL_Coord w,
                 FL_Coord h, int theta1, int thetaf, FL_COLOR col)
```

The center of the arc is the center of the bounding box specified by (x, y, w, h) and w and h specifies the major and minor axes.

To connect two points with a straight line, use the following routine

```
void fl_line(FL_Coord x1, FL_Coord y1,
             FL_Coord x2, FL_Coord y2, FL_COLOR col)
```

There is also a routine to draw a line along the diagonal of a box (to draw a horizontal line use h = 1 not 0.)

```
void fl_diagline(FL_Coord x, FL_Coord y, FL_Coord w, FL_Coord h,
                 FL_COLOR col)
```

To draw multiple connected line segments, use the following routine

```
typedef struct {short x, y} FL_POINT;

void fl_lines(FL_POINT *points, int npoint, FL_COLOR col)
```

All coordinates in points are relative to the origin of the drawable.

To change line width or style, the following convenience functions are available

```
void fl_linewidth(int lw)
void fl_linestyle(int style)
```

Use lw=0 to reset line width to the server default. Line styles can take on the following values (see *XChangeGC*(3X11))

LineSolid Solid line. Most efficient.

LineOnOffDash Dashed line. Odd-numbered dashes are drawn in the foreground color and even-numbered dashes are not drawn, thus appearing to be transparent.

LineDoubleDash Similar to LineOnOffDash, except that the even-numbered dashes are drawn with the background color (fl_bk_color()).

It is possible to change the dash patterns of `LineOnOffDash` drawing request using the following routine

```
void fl_dashedlinestyle(const char *dash, int ndashes)
```

Each element of `dash` is the length of a segment of the pattern in pixels. Dashed lines are drawn as alternating segments, each of an element in `dash`. Thus the overall length of the dash pattern, in pixels, is the sum o fall elements in `dash`. When the pattern is used up, it repeats.

By default, all lines are drawn so they overwrite the destination pixel values. It is possible to change the drawing mode so the destination pixel values play a role in the final pixel valus

```
void fl_drawmode(int mode)
```

where the supported modes are

`GXcopy` The default. Overwrite. Final value = Src;

`GXxor` Boolean exclusive-or. Useful for rubber-banding. Final value: Src `xor` dest.

`GXand` Final value: Src `and` dest.

`GXor` Final value: Src `or` dest.

`GXinvert` Final value: ~dest.

`GXnoop` Final value: dest.

There are also a number of high-level drawing routines available. To draw boxes the following routine exists. Almost any object class will use it to draw the bounding box of the object.

```
void fl_drw_box(int style, FL_Coord x,FL_Coord y,FL_Coord w,FL_Coord h,
                   FL_COLOR col, int bw)
```

Draws a box. `style` is the type of the box, e.g `FL_DOWN_BOX`. `x`, `y`, `w`, and `h`, indicate the size of the box. `c` is the color. `bw` is the width of the boundary, which typically should be given a value `obj->bw` or `FL_BOUND_WIDTH`. Note that a negative border width indicates a "softer" up box. See `DEMOS/borderwidth.c`for the visual effect of different border widths.

There is another routine that draws a frame

```
void fl_drw_frame(int style, FL_Coord x, FL_Coord y,
                  FL_Coord w, FL_Coord h, FL_COLOR col, int bw)
```

All parameters have the usual meaning except that the frame is drawn *outside* of the bounding box specified.

For drawing text there are two routines:

```
void fl_drw_text(int align, FL_Coord x, FL_Coord y, FL_Coord w,
                FL_Coord h, FL_COLOR c, int size, int style, char *str)

void fl_drw_text_beside(int align, FL_Coord x, FL_Coord y,
                       FL_Coord w, FL_Coord h, FL_COLOR c,
                       int size, int style, char *str)
```

where `align` is the alignment, namely, `FL_ALIGN_LEFT`, `FL_ALIGN_CENTER` etc. `x`, `y`, `w` and `h` indicate the bounding box, `c` is the color of the text, `size` is its size (in points), `style` is the style to be used (see Section 3.11.3 for valid styles), `str` is the string itself, possibly with embedded newlines it in. `fl_drw_text()` draws the text inside the bounding box according to the alignment request while `fl_drw_text_beside()` draws the text aligned outside the box. These two routines interpret a text string starting with the character `@` differently and draw some symbols instead. Note that `fl_drw_text()` shrinks the bounding box by 5 pixels on all sides before computing the alignment position.

The following routine can also be used to draw text and in addition, a cursor can optionally be drawn

```
void fl_drw_text_cursor(int align, FL_Coord x, FL_Coord y,
         FL_Coord w, FL_Coord h, FL_COLOR c, int size, int style,
         char *str, int FL_COLOR ccol, int pos)
```

where `ccol` is the color of the cursor and `pos` is the position of the cursor (-1 means show no cursor). This routine does not interpret the meta-character `@` nor does it shrink the bounding box in calculating the alignment position.

Given a bounding box and the size of an object (label or otherwise) to draw, the following routine can be used to obtain the starting position

```
void fl_get_align_xy(int align, int x, int y, int w, int h,
                     int obj_xsize, int obj_ysize,
                     int xmargin, int ymargin, int *xpos, int *ypos)
```

This routine works regardless if the object is to be drawn inside or outside of the bounding box specified by `x`,`y`,`w` and `h`.

An important of aspect of (re)drawing of an object is efficiency which can translate into flicker and non-responsiveness if not handled with care. For simple object like buttons or objects that do not have “movable parts", drawing efficiency is not a serious issue although you can never be too fast. For complex objects, especially those that a user can interactively change, special care should be taken.

The most important rule for efficient redrawing is don’t draw it if you don’t absolutely have to, regardless how simple the drawing is. Given the networking nature of X, simple or not depends not only on the host/server speed but also the connection. What this strategy entails is that the drawing should be broken into blocks and depending on the context, draw/updates only those parts that need to be updated.

Chapter 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 the object should turn red again and be returned to the application program. Further, the class module should keep a total count how many times the box is pushed.

The first thing to do is to define some constants in a file `colbox.h`. This file should at least contain the class number and one or more types:

```
#define FL_COLBOX        1001  /* 1001 <= Class number <= 4999 */

#define FL_NORMAL_COLBOX 0     /* The only type */
```

Note that the type must start from zero onward.

Normally it should also contain some defaults for the boxtype and label alignment etc. The include file also has to declare all the functions available for this object class. I.e., it should contain:

```
extern FL_OBJECT *fl_create_colbox(int, FL_Coord, FL_Coord, FL_Coord,
                                   FL_Coord, const char *);
extern FL_OBJECT *fl_add_colbox(int, FL_Coord, FL_Coord, FL_Coord,
                                FL_Coord, const char *);
extern int fl_get_colorbox(FL_OBJECT *);
```

Secondly we have to write a module `colbox.c` that contains the different routines. First of all we need routines to create an object of the new type and to add it to the current form. We also need to have a counter that keeps track of number of times the colbox is pushed. They would look as follows:

```
typedef struct { int counter; } SPEC;   /* no. of times pushed */

FL_OBJECT *fl_create_colbox(int type, FL_Coord x, FL_Coord y,
```

```
                            FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob;

    /* create a generic object class with an appropriate ID */
    ob = fl_make_object(FL_COLBOX,type,x,y,w,h,label,handle_colbox);

    /* initialize some members */
    ob->col1 = FL_RED;
    ob->col2 = FL_BLUE;

    /* create class specific structures and initialize */
    ob->spec = fl_calloc(1, sizeof(SPEC))
    return ob;
}

FL_OBJECT *fl_add_colbox(int type, FL_Coord x, FL_Coord y,
                        FL_Coord w,  FL_Coord h, const char *label)
{
   FL_OBJECT *ob = fl_create_colbox(type,x,y,w,h,label);
   fl_add_object(fl_current_form,ob);
   return ob;
}
```

The fields `col1` and `col2` are used to store the two colors red and blue such that the user can change them when required with the routine `fl_set_object_color()`. What remains is to write the handling routine `handle_colbox()`. It has to react to three types of events: `FL_DRAW`, `FL_PUSH` and `FL_RELEASE`. Also when the box is pushed, the counter should be incremented to keep a total count. Note that whether or not the mouse is pushed on the object is indicated in the field `ob->pushed`. Hence, when pushing and releasing the mouse the only thing that needs to be done is redrawing the object. This leads to the following piece of code:

```
static int
handle_colbox(FL_OBJECT *ob, int event, FL_Coord mx, FL_Coord my,
             int key, void *xev)
{
  switch (event) {
   case FL_DRAW:
     /* Draw box */
     if (ob->pushed)
        fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col2,ob->bw);
     else
        fl_drw_box(ob->boxtype,ob->x,ob->y,ob->w,ob->h,ob->col1,ob->bw);
     /* fall through */
   case FL_DRAWLABEL:
     /* Draw label */
     fl_draw_object_label(ob);
```

```
        return 0;
    case FL_PUSH:
        ((SPEC *)ob->spec)->counter++;
        fl_redraw_object(ob);
        return 0;
    case FL_RELEASE:
        fl_redraw_object(ob);
        return 1;
    case FL_FREEMEM:
        fl_free(((SPEC *)ob->spec));
        return 0;
    }
    return 0;
}
```

That is the whole piece of code. Of course, since structure `SPEC` is invisible outside `colbox.c`, the following routine should be provided to return the total number of times the `colbox` is pushed:

```
int fl_get_colbox(FL_OBJECT *ob)
{
    if(!ob || ob->objclass != FL_COLBOX)
    {
        fprintf(stderr, "get_colbox: Bad argument or wrong type);
        return 0;
    }
    return ((SPEC *)ob->spec)->counter;
}
```

To use it, compile it into a file `colbox.o`. An application program that wants to use the new object class simply should include `colbox.h` and link `colbox.o` when compiling the program. It can then use the routine `fl_add_colbox()` to add objects of the new type to a form.

Chapter 25

New buttons

Since button-like object is one of the most important, if not *the* most important, classes in graphical user interfaces, **Forms Library** provides, in addition to the ones explained earlier, a few more routines that make create new buttons or button-like objects even easier. These routines take care of the communication between the main module and the button handler so all new button classes created using this scheme behave consistently. Within this scheme, the programmer only has to write a drawing function that draws the button. There is no need to handle events or messages from the main module and all types of buttons, radio, pushed or normal are completely taken care of by the generic button class. Further, `fl_get_button()` and `fl_set_button()` work automatically without adding any code for them.

Forms Library provides two routines to facilitate the creation of new button object classes. One of the routines, `fl_create_generic_button()`, can be used to create a generic button that has all the properties of a real button except that this generic button does not know what the real button looks like. The other routine, `fl_add_button_class()`, provide by the **Forms Library** can be used to register a drawing routine that completes the creation of a new button.

All button or button-like object has the following instance-specific structure, defined in `forms.h`, that can be used to obtain information about the current status of the button:

```
typedef struct
{
    Pixmap pixmap;           /* for bitmap/pixmap button only        */
    Pixmap mask;             /* for bitmap/pixmap button only        */
    unsigned bits_w, bits_h; /* for bitmap/pixmap button only        */
    int val;                 /* whether pushed                       */
    int mousebut;            /* mouse button that caused the push    */
    int timdel;              /* time since last touch (TOUCH buttons)*/
    int event;               /* what event triggered the redraw      */
    long cspecl;             /* for non-generic class specific data  */
    void *cspec;              /* for non-generic class specific data  */
}
FL_BUTTON_STRUCT;
```

Of all members, only `val` and `mousebut` probably will be consulted by the drawing function.

cspecl and cspecv are useful for keeping track of class status other than those supported by the generic button (e.g., you might want to add a third color to a button for whatever purposes.) These two members are neither referenced nor changed by the generic button class.

Making this structure visible somewhat breaks the **Forms Library**'s convention of hiding the instance specific data but the convenience and consistency gained by this far outweigh the compromise on data hiding.

The basic procedures in creating a new button-like object are as follows. First, just like creating any other object classes, you have to decide on a class ID, an integer between 1001–4999 inclusive. Then write a header file so that application programs can use this new class. The header file should include the class ID definition and function prototypes specific to this new class.

After the header file is created, you will have to write C functions that create and draw the button. Also you need an interface routine to place the newly created button onto a form.

After creating the generic button, the new button class should be made known to the button driver via the following function

```
void fl_add_button_class(int objclass,
                         void (*draw)(FL_OBJECT *ob),
                         void (*cleanup)(FL_BUTTON_SPEC *));
```

where objclass is the class ID, and draw is a function that will be called to draw the button and cleanup is a function that will be called prior to destroying the button. You need a clean-up function only if the drawing routine uses cspecv field of FL_BUTTON_SPEC to hold dynamic memory allocated by the new button.

We use two examples to show how new buttons are created. The first example is taken from the button class in the **Forms Library**, that is, real working source code that implements the button class. To illustrate the entire process of creating this class, let us call this button class FL_NBUTTON.

First we create a header file to be included in an application program that uses this button class:

```
#ifndef NBUTTON_H
#define NBUTTON_H

#define FL_NBUTTON     1005

extern FL_OBJECT *fl_create_nbutton(int, FL_Coord, FL_Coord,
                                 FL_Coord, FL_Coord, const char *);
extern FL_OBJECT *fl_add_nbutton(int, FL_Coord, FL_Coord,
                                 FL_Coord, FL_Coord, const char *);
#endif
```

Now the drawing function. We use obj->col1 for the normal color of the box; obj->col2 for the color of the box when pushed. We also add an extra property that when mouse moves over the button box, it changes color. The following is the full source code that implements this:

```
typedef FL_BUTTON_STRUCT SPEC;

static void draw_nbutton(FL_OBJECT * ob)
{
    long col;

    /* box color. If pushed we use ob->col2, otherwise use ob->col1 */
    col = ((SPEC *) (ob->spec))->val ? ob->col2 : ob->col1;

    /* if mouse is on top of the button, we change the color of
     * the button to a different color. However we only do this if the
     * box has the default color.
     */
    if (ob->belowmouse && col == FL_COL1)
        col = FL_MCOL;

    /* If original button is an up_box and it is being pushed,
     * we draw a down_box. Otherwise, don't have to change
     * the boxtype
     */
    if (ob->boxtype == FL_UP_BOX && ((SPEC *) (ob->spec))->val)
       fl_drw_box(FL_DOWN_BOX, ob->x, ob->y, ob->w, ob->h, col, ob->bw);
    else
       fl_drw_box(ob->boxtype, ob->x, ob->y, ob->w, ob->h, col, ob->bw);

    /* draw the button label */
    fl_drw_object_label(ob);

    /* if the button is a return button, draw the return symbol */
    if (ob->type == FL_RETURN_BUTTON)
       fl_drw_text(0, (FL_Coord) (ob->x + ob->w - 0.8 * ob->h - 1),
                   (FL_Coord)(ob->y + 0.2 * ob->h),
                   (FL_Coord)(0.6 * ob->h),
                   (FL_Coord)(0.6 * ob->h),ob->lcol,0,0,"@returnarrow");
}
```

Since we don't use `cspecv` fieled, we don't have to write a clean-up function.

Next, following the standard procedures of the **Forms Library**, we code a separate routine that creates the new button[1]

```
FL_OBJECT *fl_create_nbutton(int type, FL_Coord x, FL_Coord y,
                             FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob;
```

[1] A separate creation routine is useful for integration into the **Form Designer**.

```
    ob = fl_create_generic_button(FL_NBUTTON, type, x, y,w,h, label);
    fl_add_button_class(FL_NBUTTON, draw_nbutton, 0);
    ob->col1 = FL_COL1;          /* normal color           */
    ob->col2 = FL_MCOL;          /* pushed color           */
    ob->align = FL_ALIGN_CENTER; /* button label placement */
}
```

You will also need a routine that adds the newly created button to a form

```
FL_OBJECT *fl_add_nbutton(int type, FL_Coord x, FL_Coord y,
                          FL_Coord w, FL_Coord h, const char *label)
{
    FL_OBJECT *ob = fl_create_nbutton(type,x,y,w,h,label);
    fl_add_object(fl_current_form, ob);
    return ob;
}
```

This concludes the creation of button class `FL_NBUTTON`.

The next example implements a button that might be added to the **Forms Library** in the future. We call this button `crossbutton`. Normally this button shows a small up box with a label on the right. When pushed, the up box becomes a down box and a small cross appears on top of it. This kind of button obviously is best used as a push button or a radio button. However, the **Forms Library** does not enforce this. It can be enforced, however, by the application program or by object class developers.

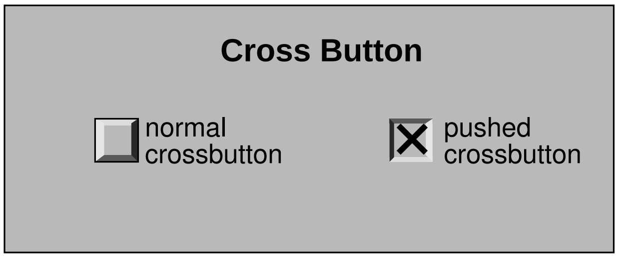

Figure 25.1: New button class

We choose to use the `ob->col1` as the color of the box and `ob->col2` as the color of the cross (remember these two colors are changeable by the application program via `fl_set_object_color()`). Note this decision on color use is somewhat arbitrary, we can easily make `ob->col2` to be the color of the button when pushed and use `ob->spec->cspecl` for the cross color (another routine `fl_set_crossbutton_crosscol(FL_OBJECT *, FL_COLOR)` should be provided to change the cross color in this case).

We start by defining the class ID and declaring the utility routine prototypes in the header file (`crossbut.h`):

```
#ifndef CROSSBUTTON_H
#define CROSSBUTTON_H

#define FL_CROSSBUTTON      2000

extern FL_OBJECT *fl_add_crossbutton(int, FL_Coord, FL_Coord,
                         FL_Coord, FL_Coord, const char *);

extern FL_OBJECT *fl_create_crossbutton(int, FL_Coord, FL_Coord,
                                    FL_Coord, FL_Coord, const char *);
```

Next we write the actual code that implements crossbutton class `crossbut.c`:

```
/*
 * routines implementing the "crossbutton" class
 */
#include "forms.h"
#include "crossbut.h"

typedef FL_BUTTON_STRUCT SPEC;

/** How to draw it */
static void draw_crossbutton(FL_OBJECT *ob)
{
    FL_Coord xx, yy, ww, hh;
    SPEC *sp = ob->spec;

    /* there is no visual change when mouse enters/leaves the box */
    if(sp->event == FL_ENTER || sp->event == FL_LEAVE)
        return;

    /* draw the bounding box first */
    fl_drw_box(ob->boxtype, ob->x, ob->y, ob->w, ob->h, ob->col1, ob->bw);

    /* draw the box that contains the cross */
    ww = hh = (0.5 * FL_min(ob->w, ob->h))  -1;
    xx = ob->x + FL_abs(ob->bw);
    yy = ob->y + (ob->h - hh)/2;

    /* if pushed, draw a down box with the cross */
    if(((SPEC *)ob->spec)->val)
    {
      fl_drw_box(FL_DOWN_BOX,xx,yy,ww,hh,ob->col1,ob->bw);
      fl_drw_text(0, xx-2, yy-2, ww+4, hh+4, ob->col2, 0, 0, "@9plus");
    }
    else
      fl_drw_box(FL_UP_BOX,xx,yy,ww,hh,ob->col1, ob->bw);
```

```
    /* label */
    if (ob->align == FL_ALIGN_CENTER)
        fl_drw_text(FL_ALIGN_LEFT, xx + ww + 2, ob->y, 0, ob->h,
                    ob->lcol, ob->lsize, ob->lstyle, ob->label);
    else
        fl_drw_text_beside(ob->align, ob->x, ob->y, ob->w, ob->h,
                    ob->lcol, ob->lsize, ob->lstyle, ob->label);

    if (ob->type == FL_RETURN_BUTTON)
       fl_drw_text(0,
                (FL_Coord)(ob->x + ob->w - 0.8 * ob->h),
                (FL_Coord)(ob->y + 0.2 * ob->h),
                (FL_Coord)(0.6 * ob->h),
                (FL_Coord)(0.6 * ob->h),ob->lcol,0,0,"@returnarrow");
}
```

This button class is somewhat different from the normal button class (FL_BUTTON) in that we enforce the appearance of a crossbutton so that an un-pushed crossbutton always has an upbox and a pushed one always has a downbox. Note that the box that contains the cross is *not* the bounding box of a crossbutton although it can be if the drawing function is coded so.

Rest of the code simply takes care of interfaces:

```
/* creation routine */
FL_OBJECT *
fl_create_crossbutton(int type, FL_Coord x, FL_Coord y, FL_Coord w,
                      FL_Coord h, const char *label)
{
     FL_OBJECT *ob;
     fl_add_button_class(FL_CROSSBUTTON, draw_crossbutton, 0);

     /* if you want to make cross button only available for
      * push or radio buttons, do it here as follows:
        if(type != FL_PUSH_BUTTON && type != FL_RADIO_BUTTON)
           type = FL_PUSH_BUTTON;
      */
     ob = fl_create_generic_button(FL_CROSSBUTTON,type,x,y,w,h,label);
     ob->boxtype = FL_NO_BOX;
     ob->col2 = FL_BLACK;   /* cross color */
     return ob;
}


/* interface routine to add a crossbutton to a form */
FL_OBJECT *
fl_add_crossbutton(int type, FL_Coord x, FL_Coord y, FL_Coord w,
```

```
                    FL_Coord h, const char *label)
{
    FL_OBJECT *ob = fl_create_crossbutton(type, x, y, w, h, label);
    fl_add_object(fl_current_form, ob);
    return ob;
}
```

The actual code is in `DEMOS/crossbut.c` and `DEMOS/crossbut.h`. An application program only needs to `#include` the header file `crossbut.h` and link with `crossbut.o` to use this new object class. There is no need to change or re-compile the **Forms Library**. Of course, if you really like the new object class, you can modify the system header file `forms.h` to include your new class header file automatically (either through inclusion at compile time or include the actual header). You can also place the object file (`crossbut.o`) in `libforms.a` if you wish. Note however, library so created may *not* be distributed.

Since the current version of **Form Designer** does not support any new object classes developed as outlined above, the best approach is to use another object class as stubs when creating a form, for example, you might want to use `checkbutton` as stubs for `crossbutton`. Once the position and size are satisfactory, generate the C-code and then manually change `checkbutton` to `crossbutton`. You probably can automate this with some scripts.

Finally there is a demo program utilizing this new button class. The program is in `newbutton.c`.

Chapter 26

Using a pre-emptive handler

Pre-emptive handlers come into being due to reasons not related to developing new classes. They are provided for the application programs to have access to the current state or event of a particular object. However, with some care, this preemptive handler can be used to override parts of the original built-in handler thus yielding a new class of objects.

26.1 The Pre-emptive and Post Object Handler

As mentioned earlier, an object module communicates with the main module via events and the agent is the event handler, which determines how an object responds to various events such as a mouse click or a key press. A pre-emptive handler is a handler which, if installed, gets called first by the main module when an event for the object occurs. The pre-emptive handler has the option to override the built-in handler by informing the main module not to call the built-in handler, thus altering the behavior of the built-in objects. The post handler, on the other hand, is called when the object handler has finished its tasks and thus does not offer the capability of overriding the built-in handler. It is much safer, however.

The API to install a pre- or post-handler for an object is as follows

```
typedef int (*FL_HANDLEPTR)(FL_OBJECT *ob, int event,
                            FL_Coord mx, FL_Coord my,
                            int key, void *raw_event);

void fl_set_object_prehandler(FL_OBJECT *ob, FL_HANDLEPTR phandler);
void fl_set_object_postehandler(FL_OBJECT *ob, FL_HANDLEPTR phandler);
```

Where `event` is the generic event in the **Forms Library**, that is, `FL_DRAW`, `FL_ENTER` etc. Parameter `mx, my` are the mouse position and `key` is the key pressed. The last parameter `raw_event` is the (cast) XEvent that caused the invocation of the pre- or post-handler. Again, not all `FL` event has corresponding `xev` and any dereferencing of `xev` should only be done after making sure it is not null.

Notice that the pre- and post-handler have the same function prototype as the built-in handler. Actually they are called with exactly the same parameters by the event dispatcher. The prehandler should return `!FL_PREEMPT` if the processing by the built-in handler should continue. A return value of `FL_PREEMPT` will prevent the dispatcher from calling the built-in handler. The post-handler is free to return anything and the return value is not used. Note that a post-handler will receive all events even if the object the post-handler is registered for does not. For example, a post-handler for a box (a static object that only receives `FL_DRAW`) receives all events.

See demo program `preemptive.c` and `xyplotall.c` for examples.

Bear in mind that modifying the built-in behavior is in general not a good idea. Using the pre-emptive handler for the purpose of “peeking", however, is quite legitimate and can be useful in some situations.

Part V

Appendices

Appendix A

Overview of main routines

In this appendix we give a brief overview of all main routines that are available. For an overview of all routines related to specific object classes see Part III.

A.1 Version Information

The header file, `forms.h`, defines three symbolic constants which you can use to conditionally compile your application. The three symbolic constants are

`FL_VERSION`	The major version number.
`FL_REVISION`	Revision number.
`FL_INCLUDE_VERSION`	Derived as `FL_VERSION` $\times$ 1000 + `FL_REVISION`

There is also a routine that can be used to obtain the library version at run time:

```
int fl_library_version(int *version, int *revision)
```

The function returns a consolidated version information, computed as `version` $\times 1000$ + `revision`. For example, for library version 1 revision 21 (1.21), the function returns a value of 1021 with `version` and `revision` (if not null) set to 1 and 21 respectively.

It is generally a good idea to check if the header and the run time library are of the same version and take appropriate actions when they are not. This is especially important for `version` < 1.

A.2 Initialization

The routine

```
Display *fl_initialize(int *argc, char *argv[], const char *appclass,
                       XrmOptionDescList app_opt, int n_app_opt)
```

initializes the **Forms Library**. It should always be called before any other calls to the **Forms Library** are made (except `fl_set_defaults()`). The meaning of the arguments are as follows

`argc,argv` Command line parameters. The application name is derived from `argv[0]` by stripping leading path names if any.

`appclass` The application class name, which typically is the generic name for all instances of this application. If no meaningful class name exists, it is typically given (or converted to if non given) as the application name with the first letter capitalized (second if the first letter is an X).

`app_opt` Specifies how to parse the application-specific resources.

`n_app_opt` Number of entries in the option list.

The `fl_initialize` function builds the resource database, calls Xlib *XrmParseCommand*(3X11) function to parse the command line, and performs other per display initialization.

All recognized options are removed from the argument list and their corresponding values set. **Forms Library** provides appropriate defaults for all options. The following are the defaults:

Options	Value type	Meaning	Default
`-debug` *level*	int	prints debug information	0
`-name` *appname*	string	changes application name	none
`-sync`	none	requests synchronous mode(debug)	false
`-display` *host:dpy*	string	specifies remote host	$DISPLAY
`-visual` *class*	string	TrueColor, pseudoColor ...	best
`-depth` *depth*	int	specifies prefered visual depth.	best
`-vid` *id*	long	specifies prefered visual ID	
`-private`	none	forces private colormap.	false
`-shared`	none	forces shared colormap.	false
`-stdcmap`	none	forces standard colormap.	false
`-double`	none	enables double buffering	false
`-bw` *width*	int	changes border width	3
`-rgamma` *gamma*	float	specifies red gamma	1.0
`-ggamma` *gamma*	float	specifies green gamma	1.0
`-bgamma` *gamma*	float	specifies blue gamma	1.0

“best" in the above table means the visual that has the most colors, which may or may not be the server default. There is a special command option `-visual Default` that sets both the visual and depth to the X server default. If a visual ID is requested, it overrides depth or visual if specified. Visual Id can also be requested programmatically (before `fl_initialize`) via the following function

```
void fl_set_visualID(long id)
```

If border width is set to a negative number, all objects appear to be softer and some people might prefer `bw -2`.

Depending on your application, **XForms** defaults may or may not be appropriate. E.g., on machines capable of 24bits visuals, **Forms Library** always selects the deeper 24bits visual. If your application only uses a limited number of colors, it would typically be faster if a visual other than 24bits is selected.

There are a couple of ways to override the default settings. You can provide an application specific resource database distributed with your program. The easiest way, however, is to set up your own program default programmatically without affecting the users' ability to override with command line options. For this, you can use the following routine *before* `fl_initialize()`:

```
void fl_set_defaults(unsigned long mask, FL_IOPT *flopt)
```

In addition to setting a preferred visual, this function can also be used to set other program defaults, such as label font size, unit of measure for form sizes etc.

See Table A.1 for a list of the masks and the members of `FL_IOPT`.

A special visual designation, `FL_DefaultVisual` and command line option equivalent `-visual Default` are provided to set the program default to the server's default visual class and depth.

If you set up your resource specifications to use class names instead of instance names, users can then list instance resources under arbitrary name that is specified with the *-name* option.

Coordinate units can be in pixels, points (1/72 inch), mm (milli-meters), cp (centi-point, i.e., 1/100 of a point) or cmm (centi-millimeter). The pre-defined designations (enums) for `coordUnit` are `FL_COORD_PIXEL`, `FL_COORD_POINT`, `FL_COORD_MM`, `FL_COORD_centiPOINT`, and `FL_COORD_centiMM`. `coordUnit` can be changed anytime, but typically you would do this prior to creating a form, presumably to make the size of the form screen resolution independent. The basic steps in doing this may look something like the following:

```
int oldcoordUnit = fl_get_coordunit();
fl_set_coordunit(FL_COORD_POINT);
fl_bgn_form(...);
/* add more objects */
fl_end_form();
fl_set_coordunit(oldcoordunit);
```

As you can see, convenience functions `fl_set_coordunit()` and `fl_get_coordunit()` are provided to change the unit of measure.

Some of the defaults are "magic" in that their exact values depend on the context or platform. For example, the underline thickness by default is 1 for normal font and 2 for bold font.

Prior to version V0.80, the origin of **XForms**'s coordinate system was at the lower-left corner of the form. The new **Form Designer** will convert the form defination file to the new coordinate system, i.e., origin at the upper-left, so no manual intervention is required. To help those who lost the .fd files or otherwise can't use the new fdesign, a compatibility function is provided

```
void fl_flip_yorigin(void)
```

Structure	Mask Name	Meaning
typedef struct {		
int debug;	FL_PDDebug	Debug level (0-5)
int depth;	FL_PDDepth	Preferred visual depth
int vclass;	FL_PDVisual	Preferred visual. TrueColor etc
int doubleBuffer	FL_PDDouble	Simulate double buffering
int buttonFontSize	FL_PDButtonFontSize	Default Button label fontsize
int menuFontSize	FL_PDMenuFontSize	Menu label fontsize
int choiceFontSize	FL_PDChoiceFontSize	Choice label and choice text fontsize
int browserFontSize	FL_PDBrowserFontSize	Browser label and text fontsize
int inputFontSize	FL_PDInputFontSize	Input label and text fontsize
int labelFontSize	FL_PDLabelFontSize	label fontsize for all other objects (box, pixmap etc.)
int pupFontSize	FL_PDPupFontSize	Fontsize for pop-ups
int privateColormap	FL_PDPrivateMap	Select private colormap if appropriate
int sharedColormap	FL_PDSharedMap	Force shared colormap always
int standardColormap	FL_PDStandardMap	Force standard colormap
int leftScrollBar	FL_PDLeftScrollBar	Placement of scrollbar in browser
int ulThickness	FL_PDULThickness	Underline thickness
int ulPropWidth	FL_PDULPropWidth	Underline width. 0 for const. width
int coordUnit	FL_PDCoordUnit	Unit of measure: pixel, mm, point
int borderWidth	FL_PDBorderWidth	Height of an object
} FL_IOPT;		

Table A.1: `FL_IOPT` structure

Note however, this function must be called prior to `fl_initialize` and is a no-op after that.

For proportional font, substituting tabs with spaces is not always appropriate because this most likely will fail to align text properly. Instead, a tab is treated as an absolute measure of distance, in pixels, and a tab stop will always end at multiples of this distance. Application program can adjust this distance by setting the tab stops using the following routine

```
void fl_set_tabstop(const char *s)
```

where `s` is a string whose width in pixels is to be used as the tab length. The font used to calculate the width is the same font that is used to render the string in which the tab is embedded. The default is `s` = `"aaaaaaaa", i.e., eight \verb'a'+s;`

Before we proceed further, some comments about double buffering are in order. Since Xlib does not support double buffering, **Forms Library** simulates this functionality with pixmap bit-blting. In practice, the effect is hardly distinguishable from double buffering and performance is on par with multi-buffering extensions (It is slower than drawing into a window directly on most workstations however). Bear in mind that pixmap can be resource hungry, so use this option with discretion.

In addition to using double buffering throughout an application, it is also possible to use double buffering on a per-form or per-object basis by using the following routines:

```
void fl_set_form_dblbuffer(FL_FORM *form, int yes)
void fl_set_object_dblbuffer(FL_OBJECT *obj, int yes)
```

Currently double buffering requests for objects having a non-square box are ignored.

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,const char *name)
```

See section 3.11.3 for details. A redraw of all forms is required to actually see the change.

Since the dimension of an object is typically given in pixels, depending on the server resolution and the font used, this can lead to unsatisfactory user interfaces. For example, a button designed to (just) contain a label in a 10pt font on a 75 DPI monitor will have the label overflow the button on a 100DPI monitor. This comes about because a character in a 10pt font with 75DPI resolution may have 10 pixels while the same character in the same 10 pt font with 100DPI resolution may have 14 pixels. Thus when designing the interfaces, leave a few pixels extra for the object. Or use a resolution independent unit, such as point, or centi-point etc.

Using a resolution independent unit for the object size should solve the font problems theoretically. In practice, this approach may still prove to be vulnerable. This comes about because of the discreteness of both the font resolution and the monitor/server resolutions. The standard X fonts only come in two discrete resolutions, 75 DPI and 100 DPI. Due to the variations in monitor resolutions, the same theoretically same sized font, say a 10pt font, can vary in sizes up to 30% depending on the server (rendering a font on a 80DPI monitor will cause error in sizes regardless if 75 or 100DPI font is used.) This has not even taken into account the fact that a surprisingly number of systems have wrong font paths (i.e. a 90DPI monitor using 75DPI fonts etc).

XForms works around these problems by providing a function that can be used to adjust the object size according to the actual fonts loaded

```
void fl_fit_object_label(FL_OBJECT *obj, FL_Coord hm, FL_Coord vm);
```

where `hm` and `vm` are, respectively, the horizontal and vertical margins to leave on each side of the object. This function works by computing the object label size and compares to the object size. If the label does not fit inside the object with the given margin, the entire form the object is on is scaled so the object label fits. In scaling the form, all gravity specification is ignored. This function will not shrink a form. You can use this function on as many objects as you choose. Of course the object has to have a label inside the object for this function to work.

In some situations **Forms Library** may modify some of the server defaults. All modified defaults are restored as early as possible by the main loop and in general when an application exits, all server defaults are restored. The only exception is that when exiting from a callback that is activated by shortcuts. Thus it is recommanded that the cleanup routine `fl_finish()` be called prior to exiting an application or register it via *atexit*(3)

```
void fl_finish(void)
```

In addition to restoring all server defaults, `fl_finish()` will also shut down the connection.

A.3 Creating forms

```
FL_FORM *fl_bgn_form(int type,FL_Coord w,FL_Coord h)
```

Starts the definition of a form. `type` is the type of the box that is used as a background. `w` and `h` give the width and height of the form. The function returns a pointer to the form created.

```
void fl_end_form()
```

End the definition of a form.. Between these two calls, various objects, including group of objects, are added to the form.

```
FL_OBJECT *fl_bgn_group()
```

Begin the definition of a group of objects inside the form. It returns a pointer to the group. Groups should never be nested.

```
FL_OBJECT * fl_end_group(void)
```

Ends the definition of a group.

Groups are useful for two reasons. First of all, it is possible to hide or deactivate groups of objects. This is often very handy to dynamically change the appearance of a form depending on the context or selected options. In addition, it can also be used as a shortcut to set some particular attributes of several objects. It is not uncommon that you want several objects to maintain their relative positioning upon form resizing. This requires to set the gravity for each object. If these objects are placed inside a group, setting the gravity attributes of the group would suffice.

The second reason for using groups is for radio buttons. Radio buttons are considered related only if they belong to the same group. Using groups is the only way to place unrelated groups of radio buttons on a single form without interference from each other.

```
void fl_addto_group(FL_OBJECT *group)
```

reopens a group for adding more objects to it. Any new objects added are appended at the end of the group.

```
void fl_addto_form(FL_FORM *form)
```

Reopens a form for adding objects to it.

```
void fl_delete_object(FL_OBJECT *obj)
```

Removes an object from the form it is in.

```
void fl_free_object(FL_OBJECT *obj)
```

Frees the memory for an object. (Object should be deleted first.) An object after being freed should not be referenced.

```
void fl_free_form(FL_FORM *form)
```

Frees the memory for a form, together with all its objects. The form should not be visible.

A.4 Doing interaction

```
long fl_show_form(FL_FORM *form,int place,int border,const char *title)
```

Displays a form on the screen. `place` controls the position and size of the form. `border` indicates whether a border (window manager's decoration) should be drawn around the form. In this case `title` is the name of the window. The routine returns the window identifier of the form. For resource and identification purposes, the form name is taken to be the title with space removed and the first character lower-cased. E.g., if a form has a title `Foo Bar` the form name is derived as `fooBar`.

There are variations on the `border` requests:

`FL_FULLBORDER`	full border with title showing.
`FL_TRANSIENT`	border possibly with less decoration.
`FL_NOBORDER`	no border at all.

Multiple forms can be displayed at the same moment.

Note that `FL_NOBORDER` might have adverse effect on keyboard focus and is not very friendly to other applications (it is close to impossible to move a form that has no border). Thus use this feature with discretion. The only situation where `FL_NOBORDER` is appropriate is for automated demonstration suites or when the application program *must* obtain an input or a mouse click from the user, and even then all other forms should be deactivated while a borderless form is active. For almost all situations where the application must demand an action from the user, `FL_TRANSIENT` is preferred. Also note you can't iconify a form that has no border and under most window managers, `FL_TRANSIENT` form can't be iconified either.

On additional property (under almost all window managers) of a transient window is that it will stay on top of the main form, which the application program can designate using

```
void fl_set_app_mainform(FL_FORM *form)
```

By default, the main form is set automatically by the library to the first full-bordered form shown.

To obtain the current main form, use the following routine

```
FL_FORM *fl_get_app_mainform(void)
```

In some situations, either because the concept of an application main form does not apply (for example, an application might have multiple full-bordered windows), or under some (buggy) window managers, the designation of a main form may cause stacking order problems. To workaround these, the following routine can be used to disable the designation of a main form (before any full-bordered form is shown)

```
void fl_set_app_nomainform(int flag)
```

with a true `flag`

All visible forms will have the following properties set

```
WM_CLASS
WM_CLIENT_MACHINE
WM_NAME
```

In addition, the first full border form will have the `WM_COMMAND` property set and is by default the application main form.

The application program can raise a form to the top of the screen so no other forms obscure it by using the following routine

```
void fl_raise_form(FL_FORM *form)
```

Similar routine exists that lowers a form to the bottom of the stack

```
void fl_lower_form(FL_FORM *form)
```

When placing a form on the screen using place `FL_PLACE_GEOMETRY` the position and size can be set using the routines

```
void fl_set_form_position(FL_FORM *form, FL_Coord x, FL_Coord y)

void fl_set_form_size(FL_FORM *form, FL_Coord w, FL_Coord h)

void fl_scale_form(FL_FORM *form, double xsc, double ysc)
```

Where the last routine scales with a factor with respect to the current size. These routines can also be used when the form is visible.

If interactive resizing is to be allowed, (e.g., form shown using `FL_PLACE_FREE`), it can be useful to limit the range the size of a form can take. To this end, the following functions are available

```
void fl_set_form_minsize(FL_FORM *form, FL_Coord minw, FL_Coord minh);

void fl_set_form_maxsize(FL_FORM *form, FL_Coord maxw, FL_Coord maxh);
```

Although these two routines can be used before or after a form becomes visible, not all window managers honor such requests once the window is visible.

To set or change the icon shown when a form is iconified, use the following routine

```
void fl_set_form_icon(FL_FORM *form, Pixmap icon, Pixmap mask)
```

where `icon` can be any valid Pixmap ID. (See Sections 13.4 and 13.5 for some of the routines that can be used to create Pixmaps) Note that the previous icon if not freed or modified in anyway.

If for any reason, you would like to change the form title after it is shown, the following call can be used (this will also change the icon title)

```
void fl_set_form_title(FL_FORM *form, const char *name)
```

The routine

```
void fl_hide_form(FL_FORM *form)
```

hides the particular form, i.e., closes its window and all subwindows, if any.

```
FL_OBJECT *fl_do_forms(void)
```

Does the interaction with the currently displayed forms. The routine ends when the state of some object changes. A pointer to this object is returned.

```
FL_OBJECT *fl_check_forms(void)
```

Second way of doing interaction with the currently displayed forms. The routine returns immediately `NULL` unless the state of some object changes in which case a pointer to this object is returned.

```
FL_OBJECT *fl_do_only_forms(void)

FL_OBJECT *fl_check_only_forms(void)
```

Same as `fl_{do|check}_forms` except that these routines do not handle user events.

```
void fl_activate_form(FL_FORM *form)
```

Activates a form for user interaction.

```
void fl_deactivate_form(FL_FORM *form)
```

Deactivates a form to stop user interaction with it.

```
void fl_deactivate_all_forms(void)
```

```
void fl_activate_all_forms(void)
```

Activates or deactivates all forms to stop user interaction with them.

You can also register for a form callbacks that are invoked whenever the activation status of the form is changed:

```
typedef void (*FL_FORM_ATACTIVATE)(FL_FORM *, void *);

FL_FORM_ACTIVATE fl_set_form_atactivate(FL_FORM *form,
                        FL_FORM_ATACTIVATE callback, void *data);

FL_FORM_ACTIVATE fl_set_form_atdeactivate(FL_FORM *form,
                        FL_FORM_ATACTIVATE callback, void *data);
```

```
void fl_activate_object(FL_OBJECT *obj)
```

Activates an object for user interaction.

```
void fl_deactivate_object(FL_OBJECT *obj)
```

Deactivates an object to stop user interaction with it.

```
void fl_redraw_object(FL_OBJECT *obj)
```

This routine redraws the particular object. If `obj` is a group it redraws the complete group. Normally you should never need this routine because all library routines take care of redrawing but there might be situations in which a redraw is required.

```
void fl_redraw_form(FL_FORM *form)
```

Redraws an entire form.

A.5 Setting attributes

A number of general routines are available for setting attributes. Unless stated otherwise, all attributes altering routines affect the appearance or geometry of the object immediately if the object is visible.

```
void fl_set_object_color(FL_OBJECT *obj, int col1, int col2)
```

Sets the two colors that influence the appearance of the object.

attributes:

```
void fl_set_object_boxtype(FL_OBJECT *obj, int boxtype)
```

Changes the shape of the bounding box of the object.

```
void fl_set_object_position(FL_OBJECT *ob, FL_Coord x, FL_Cood y)
```

sets a new position for the object. If the object is visible, it is moved to the new location.

```
void fl_set_object_size(FL_OBJECT *ob, FL_Coord w, FL_Coord h)
```

changes the object size while keeping the upper-left corner of the bounding box unchanged.

```
void fl_set_object_geometry(FL_OBJECT *ob, FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h)
```

sets both the position of the size of an object.

changes the size of the object. If the object is visible, the change is shown immediately.

```
void fl_set_object_lcol(FL_OBJECT *obj, int lcol)
```

```
void fl_set_object_lsize(FL_OBJECT *obj, int lsize)
```

```
void fl_set_object_lstyle(FL_OBJECT *obj, int lstyle)
```

```
void fl_set_object_lalign(FL_OBJECT *obj, int align)
```

```
void fl_set_object_label(FL_OBJECT *obj, const char *label)
```

These routines set the color, size, style, alignment and text of the label of the object.

```
void fl_set_object_resize(FL_OBJECT *obj, unsigned howresize)
```

```
void fl_set_object_gravity(FL_OBJECT *obj,
                           unsigned NWgravity, unsigned SEgravity)
```

If you change many attributes of a single object or many objects in a visible form, the changed object is redrawn after each change. To avoid this, put the changes between calls to

```
void fl_freeze_form(FL_FORM *form)
```

and

```
void fl_unfreeze_form(FL_FORM *form)
```

There are also routines that influence the way events are dispatched. These routines are provided mainly to facilitate the development of (unusual) new objects where attributes need to be changed on the fly. These routines should not be used on the built-in ones.

To enable or disable an object to receive the `FL_STEP` event, use the following routine

```
void fl_set_object_automatic(FL_OBJECT *obj, int flag)
```

To enable or disable an object to receive the `FL_DBLCLICK` event, use the following routine

```
void fl_set_object_dblclick(FL_OBJECT *obj, int timeout)
```

```
void fl_show_object(FL_OBJECT *obj)
```

Makes the object, or the group if `obj` is a group, visible.

```
void fl_hide_object(FL_OBJECT *obj)
```

makes the object or group invisible.

```
void fl_trigger_object(FL_OBJECT *obj);
```

returns `obj` to the application program or calls `obj`'s callback if one exists.

```
void fl_set_focus_object(FL_FORM *form,FL_OBJECT *obj)
```

Set the input focus in form `form` onto object `obj`.

The routine

```
void fl_set_object_callback(FL_OBJECT *obj,
                            void (*callback)(FL_OBJECT *, long),
                            long argument)
```

binds a callback routine to an object.

To invoke the callback manually (as opposed to invocation by the main loop), use the following function

```
void fl_call_object_callback(FL_OBJECT *obj)
```

If the object `obj` does not have a callback associated with it, this call has not effect.

```
void fl_set_form_callback(FL_FORM *form,
                   void (*callback)(FL_OBJECT *, void *), void *data)
```

Binds a callback routine to an entire form.

```
void fl_set_event_callback(void (*callback)(void *xevent, void *data))
```

Sets up a callback routine for all user events.

It is possible to set up callback functions on a per window/event basis using the following routines

```
typedef int (*FL_APPEVENT_CB)(XEvent *xev, void *user_data);

FL_APPEVENT_CB fl_add_event_callback(Window win, int xevent_type,
                           FL_APPEVENT_CB callback, void *user_data);

void fl_remove_event_callback(Window win, int xevent_type)
```

These functions manipulate event callback functions for the window specified and will be called when `xevent_type` is pending for the window. If `xevent_type` is zero, it signifies a callback for *all* event for window `win`. Note that **Forms Library** does not solicit any event for the caller, i.e., **Forms Library** assumes the caller opens the window and solicits all events before calling these routines. To let **Forms Library** handle event solicitation, the following function may be used

```
void fl_activate_event_callback(Window win);
```

It is sometimes useful to obtain the last event from within a callback function, e.g., to implement different functionalities depending on which button triggers the callback. For this, the following routine can be used from within a callback function

```
const XEvent *fl_last_event(void)
```

A.6 Signals

The application program can elect to handle the receipt of a signal by registering a callback function that gets called when the signal is raised and caught

```
typedef void (*FL_SIGNAL_HANDLER)(int, void *);
void fl_add_signal_callback(int signal, FL_SIGNAL_HANDLER sh, void *data);
```

Only one callback per signal is permitted.

By default, `fl_add_signal_callback` will store the callback function and initiate mechanism for the OS to deliver the signal when it occurs. When the signal is received by the library, the main loop will invoke the registered callback function when it is appropriate to do so. The callback function can make use of all **XForms**'s functions as well as Xlib functions as if there were re-entrant. Further, a signal callback so registered is persistent and will cease to function only when explicitly removed.

It is very simple to use this routine. For example, to prevent a program from exiting prematurely due to interrupts, code fragment similar to the following can be used:

```
void clean_up(int signum, void *data)
{
     /* clean up, of course */
}

/* and somewhere after fl_initialize */

fl_add_signal_callback(SIGINT, clean_up, &mydata);
```

After this, whenver interrupt is detected, `clean_up` is called.

To remove a signal callback, the following routine should be used

```
void  fl_remove_signal_callback(int signal);
```

There are limitations with the default behavior outlined above. For example, on some platforms, there is no blocking of signals of any kind while handling a signal. In addition, use of `fl_add_signal_callback` prevents the application program from using any, potentially more flexible, signal handling routines on some platforms.

In light of these limitations, provisions are made so an application program may choose to take over the initial signal handling setup and receipt via various system dependent methods (*sigaction*(2) for example).

To change the default behavior of built-in signal facilities, the following routine should be called prior to any use of `fl_add_signal_callback` with a true `flag`:

```
void fl_app_signal_direct(int flag)
```

After this call, `fl_add_signal_callback` will not initiate any actions to receive a signal. The application program should handle the receipt and blocking of a signal (via, *signal*(2), *sigaction*2, *sigprocmask*2 etc.) When the signal is received by the application program, it should call the following routine to inform the main loop of the delivery of the signal `signum`

```
void fl_signal_caught(int signum);
```

This routine is the only one in the library that can be safely called from within a direct application signal handler. If multiple invokation of `fl_signal_caught` occurs before the main loop is able to call the registered callback, the callback is called only once.

Appendix B

Resources for Forms Library

Managing resources is an important part of programming with X. Typical X programs use extensive resource database/management to customize their appearances. With the help of **Form Designer**, there is little or no need to specify any resources for the default appearance of an application written using the **Forms Library**. Because of this, complete resource support is somewhat a low-priority task and currently only minimal support is available. Nevertheless, more complete and useful resource management system specific to the **Forms Library** can be implemented using the services provided by the **XForms**.

B.1 Current Support

At the moment, all built-in **XForms** resources have a top level class name `XForm` and a resource name `xform`. Because of this incomplete specification, most of the current resources are "global", in the sense that they affect all form windows. Eventually all resources will be fully resolved, e.g., to specify attribute `foo` of form `formName`, the resource name can be `appName.formName.foo` instead of (the current incomplete) `appName.xform.foo`.

The argument `app_opt` in `fl_initialize()` is a table of structures listing your application's command line options. The structure is defined as follows

```
typedef struct
{
    char *option;
    char *specifier;
    XrmOptionKind argKind;
    void *value;
} XrmOptionDescList, FL_CMD_OPT;
```

See *XrmGetResource*(3X11) for details.

After the initialization routine is called, all command line arguments, both **XForms** built-in and application specific ones, are removed from `argc` and `argv` and parsed into a standard XResources database. To read your application specific options, follow `fl_initialize()` with the following routine

```
int fl_get_app_resources(FL_resource *resource, int nresources);
```

Here `resource` is a table containing application specific resources in the following format:

```
typedef struct
{
    char *res_name;    /* resource name without application name    */
    char *res_class;   /* resource class                            */
    FL_RTYPE type;     /* C type of the variable                    */
    void  *var         /* variable that will hold the value         */
    char *defval;      /* default value in string form              */
    int nbytes;        /* buffer size for string var.               */
} FL_RESOURCE;
```

and the resource type `FL_RTYPE type` is one of the following

```
FL_SHORT          for short   variable
FL_BOOL           for boolean variable (int)
FL_INT            for int     variable
FL_LONG           for long    variable
FL_FLOAT          for float   variable
FL_STRING         for char [] variable
```

Note that the variable for `FL_BOOL` must be of type `int`. It differs from `FL_INT` only in the way the resources are converted, not in the way their values are stored. A boolean variable is considered to be true (1) if any one of `True, true, Yes, yes, On, on,` or `1` is specified as its value. For string variables, the length for the destination buffer must be specified.

`fl_get_app_resources()` simply looks up all entries specified in `FL_resource` structure in all databases after prefixing the resource name with the application name, which can be the new name introduced by the `-name` option.

Summarized below are the currently recognized **Forms Library** built-in resources:

Resource Name	Class	Type	Default
rgamma	Gamma	float	1.0
ggamma	Gamma	float	1.0
bgamma	Gamma	float	1.0
visual	Visual	string	best
depth	Depth	int	best
doubleBuffer	DoubleBuffer	bool	false
privateColormap	PrivateColormap	bool	false
standardColormap	StandardColormap	bool	false
sharedColormap	SharedColormap	bool	false
pupFontSize	PupFontSize	int	12pt
buttonFontSize	FontSize	int	10pt
sliderFontSize	FontSize	int	10pt
inputFontSize	FontSize	int	10pt
browserFontSize	FontSize	int	10pt
menuFontSize	FontSize	int	10pt
choiceFontSize	FontSize	int	10pt
ulPropWidth	ULPropWidth	bool	true
ulThickness	ULThickness	int	1
leftScrollBar	LeftScrollBar	bool	false
coordUnit	CoordUnit	string	pixel
borderWidth	BorderWidth	int	3

Again, “best" means that the **Forms Library** by default selects a visual that has the most depth.

By default, resource files are read and merged in the order as suggested by X11 R5 as follows:

- `/usr/lib/X11/app-defaults/<AppClassName>`
- `$XAPPRLESDIR/<AppClassName>`
- `RESOURCE_MANAGER` property as set using *xrdb*
  if `RESOURCE_MANAGER` is empty, `~/.Xdefaults`
- `$XENVIRONMENT`
  if `$XENVIORONMENT` is empty, `~/.Xdefaults`-*hostname*
- command line options

All options set via resources may *not* be the final values used because resource settings are applied at the time object/form is created, thus any modifications after that override the resource settings. For example, `buttonLabelSize`, if set, is applied at the time the button is created (`fl_add_button()`). Thus altering the size after the button is created via `fl_set_object_lsize()` overrides whatever is set by the resource database.

To run your application in `PseudoColor` with a depth of 8 and a thick underline, specify the following resources

```
appname*visual:         PseudoColor
appname*depth:          8
appname*ulThickness:    2
```

Since resources based on a form by form basis are yet to be implemented, there is no point specifying anything more specific although `appname.XForm.depth` etc. would work correctly.

An example

Let us assume that you have an application named `myapp`, and it accepts the options `-foo` *level* and `-bar`, and a filename. The proper way to initialize the **Forms Library** is as follows

```
FL_CMD_OPT cmdopt[] =
{
   {"-foo", "*.foo", XrmoptionSepArg, 0},
   {"-bar", ".bar", XrmoptionNoArg, "True"},
};

int foolevel, ifbar;
int deftrue;    /* can only be set thru resources */

FL_resource res[] =
{
    {"foo", "FooCLASS", FL_INT, &foolevel, "0"},
    {"bar", "BarCLASS",FL_BOOL, &ifbar,"0"},
    {"deftrue", "Whatever",FL_BOOL, &deftrue,"1"}
};

int
main(int argc, char *argv[])
{
   fl_initialize(&argc, argv ,"MyappClass", cmdopt, 2);
   fl_get_app_resources(res, 3);
   if(argc == 1)   /* missing filename */
      fprintf(stderr,"Usage %s: [-foo level][-bar] filename\n","myapp");
   /* rest of the program */
}
```

After this, both `foolevel` and `ifbar` are set either through resource files or command line options with the command line options overriding those set in the resource file. In case neither the command line nor the resource file specified the options, the default value string is converted.

There is another routine, the resource routine of the lowest level in **XForms**, might be useful if a quick& dirty option needs to be read:

```
const char *fl_get_resource(const char *res_name, const char *res_class,
                       FL_RTYPE type, char *defval, void *val, int nbytes)
```

`res_name` and `res_class` here must be complete resource specifications (minus the appliation name) and should not contain wildcard of any kind. The resource will be converted according to the `type` and result stored in `type`. *nbytes* is used only if the resource type is `FL_STRING`. The function returns the string representation of the resource value. If `type` is passed a value `FL_NONE`, the resource is not converted and the pointer `val` is not referenced.

There is also a routine that allows the application program to set resources programatically

```
void fl_set_resource(const char *string, const char *value)
```

where `string` and `value` are a resource-value pair. The string can be a fully qualified resource name (minus the application name) or a resource class.

Routines `fl_set_resource` and `fl_get_resource` can be used to store and retrieve arbitary strings and values and may be useful to pass data around.

B.2 Going Further

It is possible to implement your own form/object specific resources management system using the services mentioned above. For example, to implement a user-configurable form size, code similiar to the following can be used, assuming the form is named “myform"

```
 struct fsize { int width, height; } myformsize;

 FL_resource res[] =
 {
    {"myform.width", "XForm.width", FL_INT, &(myform.width),   "150"},
    {"myform.height","XForm.height", FL_INT, &(myform.height), "150"},
 };

 fl_initialize(&argc, argv, app_class, 0, 0);
 fl_get_app_resources(res,2);

/* create the forms */
 myform = fl_bgn_form (myformsize.width, myformsize.height,.....);
```

Or (more realistically) you create the form first using `fdesign` and then scale it before it is shown:

```
fl_initialize(&argc, argv, app_class, 0, 0);
fl_get_app_resources(res,2);
/*create_all_forms here */
fl_set_form_size(myform, mysformsize.width, myformsize.height);
fl_show_form(myform, ...);
```

Eventully form geometry and other things might be done via **XForms** internal routines, it is recommended that you name your form to be the form title with all spaces removed and first letter lower-cased, i.e., if a form is shown with a label `Foo Bar`, the name of the form should be `fooBar`.

Appendix C

Dirty Tricks

This chapter describes some of the routines that may be used in special situations where more power or flexibility from **Forms Library** is needed. These routines are classified as “dirty tricks" either because they can easily mess up the normal operation of **Forms Library** or they depend on internal information that might change in the future. Thus whenever possible, try *not* to use these routines.

C.1 Interaction

C.1.1 Form Event

It is possible to by-pass the form event processing entirely by setting a “raw callback” that sits between the event reading and dispatching stage, thus a sneak preview can be implemented and optionally consume the event before the internal form processing machinery gets to it.

Use the following routines to register such a preemptive processing routine

```
typedef int (*FL_RAW_CALLBACK)(FL_FORM *, void *xevent);

FL_RAW_CALL_BACK fl_register_raw_callback(FL_FORM *form,
                        unsigned long mask,
                        FL_RAW_CALLBACK callback);
```

where `mask` is the event mask you are interested in (same as XEvent mask). The function returns the old handler for the event.

Currently only handlers for the following events are supported

- `KeyPressMask` and `KeyReleaseMask`
- `ButtonPressMask` and `ButtonReleaseMask`
- `EnterWindowMask` and `LeaveWindowMask`

- `ButtonMotionMask` and `PointerMotionMask`

Further there is only one handler for each event pair, (e.g., `ButtonPress` and `ButtonRelease`), thus you can't have two separate handlers for each pair although it is okay to register a handler only for one of them (almost always a mistake) if you know what you're doing. If you register a single handler for more than one pair of events, e.g., setting `mask` to `KeyPressMask|ButtonPressMask`, the returned old handler is random.

A special constant, `FL_ALL_EVENT`, is defined so that the handler registered will received *all* events that are selected. To select events, use `fl_addto_selected_xevent()`.

Once an event handler is registered and the event is detected, then instead of doing the default processing by the dispatcher, the registered handler is invoked. The handler must return `FL_PREEMPT` if the event is gobbled up (consumed) and 0 otherwise so that the internal process can continue. See `minput2.c` for an example.

C.1.2 Object Events

Just as you can by-pass the internal event procesing for a particular form, you can also do so for an object. Unlike in raw callbacks, you can not select individual events.

The mechnism provided is via the registration of a pre-handler for an object. The pre-handler will be called before the built-in object handler. By electing to handle some of the events, a pre-handler can, in effect, replace part of the built-in handler.

Chapter 26.1 has already discussed the API in detail, here we just repeat the discussion for completeness as any use of preemptive handler is considered “dirty tricks".

To register a pre-handler, use the following routine

```
typedef int (*FL_HANDLEPTR)(FL_OBJECT *ob, int event,
                            FL_Coord mx, FL_Coord my,
                            int key, void *raw_event);

void fl_set_object_prehandler(FL_OBJECT *, FL_HANDLEPTR prehandler);
```

Where `event` is the generic event in the **Forms Library**, that is, `FL_DRAW`, `FL_ENTER` etc. Parameter `mx, my` are the mouse position and `key` is the key pressed. The last parameter `raw_event` is the (cast) XEvent that caused the invocation of the pre-handler.

Notice that the pre-handler has the same function prototype as the built-in handler. Actually they are called with the same exact parameters by the event dispatcher. The prehandler should return 0 if the processing by the built-in handler shoud continue. A return value of `FL_PREEMPT` will prevent the dispatcher from calling the built-in handler.

See demo program `preemptive.c` for an example.

Similar mechnism exists for registering a post-handler, i.e., a handler invoked after the built-in handler finishes. Whenever possible a post-handler should be used instead of a pre-handler.

C.2 Windowing Support

Some of these routines are used internally by the **Forms Library** as an attempt to localize window system dependencies and may be of some general use. Be warned that these routines may subject to frequent changes, both in their API and/or functionality.

You can create and show a window with the following routines

```
Window fl_wincreate(const char *name)
Window fl_winshow(Window win);
```

where parameter `win` in `fl_winshow` is the window ID returned by `fl_wincreate`. Between the creation and showing of the window, other attributes of the window can be set. Note a window so opened is always a top level window and uses all the **Forms Library**'f defaults (visual, depth etc.)

It is sometimes more convenient to create and show a window directly

```
Window fl_winopen(const char *name)
```

This will open a (top-level) window with a title `name`. A window so opened can be drawn into as soon as the function returns, i.e., `fl_winopen()` waits until the window is ready to be drawn into.

The newly opened window will have the following default attributes

`event_mask` `ExposureMask`, `KeyPressMask`, `KeyReleaseMask`, `ButtonPressMask`, `ButtonReleaseMask`, `OwnerGrabButtonMask`, `ButtonMotionMask`, `PointerMotionMask`, `PointerMotionHintMask`, `StructureNotifyMask`

`backing_store` `fl_cntl.backingStore`

`class` `InputOutput`

`visual` same as **Forms Library**'s default.

`colormap` same as **Forms Library**'s default.

By default, the newly opened window will have a size of 320 by 200 pixels and has no other constraints. You can modify the default or constraints using the following routines prior to calling `fl_winopen()`:

```
void fl_initial_winsize(FL_Coord w, FL_Coord h)
void fl_winsize(FL_Coord w, FL_Coord h)
```

These two routines set the preferred window size. `w` and `h` are the width and height of the window in pixels. `fl_winsize()` in addition will make the window non-resizable (You can still resize the window programmatically) by setting the minimum and maximum window size to the requested size via `WMHints`. The effect of a window having this property is that it can't be interactively resized (provided the window manager cooperates).

It is sometimes desirable to have a window that is resizable but only within a useful range. To set such a constraint, use the following functions:

```
void fl_winminsize(Window window, FL_Coord minw, FL_Coord minh)
void fl_winmaxsize(Window window, FL_Corrd maxw, FL_Coord maxh)
```

These two routines can also be used after a window becomes visible. For windows to be created/opened, use 0 for the `window` parameter. For example, if we want to open a window of 640 by 480 pixels, and have it remain resizeable but within a permitted range, code similar to the following can be used:

```
fl_initial_winsize(640,480);
fl_winminsize(0, 100,100);
fl_winmaxsize(0, 1024,768)
win = fl_winopen("MyWin");
```

In addition to window size preference, you can also set preferred position of a window to be opened:

```
void fl_winposition(FL_Coord x, FL_Coord y)
```

where `x` and `y` are the coordinates of the upper-left corner of the window relative to the root window.

Or you can set the geometry in one function call

```
void fl_initial_wingeometry(FL_Coord x, FL_Coord y,
                            FL_Coord w, FL_Coord h)

void fl_wingeometry(FL_Coord x, FL_Coord y,
                    FL_Coord w, FL_Coord h)
```

Again, the `fl_wingeometry` will deny later interactive resizing.

There are other routines that can be used to change other aspects of the window to be created

```
fl_winaspect(Window win, FL_Coord x, FL_Coord y)
```

This will set the aspect ratio of the window in later interactive resizing.

To change the window title (and its associated icon title), use the following routine

```
void fl_wintitle(Window win, const char *title)
```

To install an icon for the window, use the following routine

```
void fl_winicon(Window win, Pixmap icon, Pixmap mask)
```

You can suppress the window manager's decoration or make the window a transient one by using the following routines

```
void fl_noborder(void)
void fl_transient(void)
```

You can also set the background of the window to a certain color using the following call

```
void fl_winbackground(Window win, unsigned long pixel)
```

It is possible to set the change size of a window to some discrete steps:

```
void fl_winstepsize(Window win, int xunit, int yunit)
```

where `xunit` and `yunit` are the amount of changes per unit in the x- and y- directions respectively. Note that this only applies to interactive resizing.

To change constraints (size and aspect ratio) on an active window, you can use the following routine

```
void fl_reset_winconstraints(Window win)
```

The following routines are available to get information about an active window `win`:

```
void fl_get_winsize(Window win, FL_Coord *w, FL_Coord *h)
void fl_get_winorigin(Window win, FL_Coord *x, FL_Coord *y)
void fl_get_wingeometry(Window win, FL_Coord *x, FL_Coord *y,
                        FL_Coord *w, FL_Coord *h)
```

All units are in pixels. Origin of a window is measured from the upper left corner of the root window.

To change the size of window programmatically, the following routine is available:

```
int fl_winresize(Window win, FL_Coord neww, FL_Coord newh)
```

In addition to resizing the window, this routine will keep the original constraints. For example, if a window was not permitted to be interactively resized, after the resize, it remains unresizeable. Resizing is done by keeping the origin constant relative to the root window.

To move a window without resizing it, use the following call

```
void fl_winmove(Window win, FL_Coord newx, FL_Coord newy)
```

To move and resize a window, use the following routine

```
void fl_winreshape(Window win, FL_Coord newx, FL_Coord newy,
                   FL_Coord neww, FL_Coord newh)
```

To make a window invisible, use the following routine

```
void fl_winhide(Window win)
```

A hidden window can be shown again later using `fl_winshow`.

To hide and destroy a window, use the following calls

```
void fl_winclose(Window win)
```

There will be no events generated from `fl_winclose`, i.e., the function waits and eats all events for window `win`. In addition, this routine also removes all callbacks associated with the closed window.

The following routine can be used to check if a window is valid or not

```
int fl_winisvalid(Window win)
```

Note that excessive use of this function may negatively impact performance.

XForms also provides a convenience function to change the cursor shapes

```
void fl_set_cursor(Window win, int name)
```

where `win` must be a valid window identifier and `name` is one of the symbolic cursor names (shapes) defined by standard X. These symbolic names are defined in `<X11/cursorfont.h>`. For example, to set a watch-shaped cursor for form `form`, the following call may be made

```
void fl_set_cursor(form->window,  XC_watch);
```

Depending on the structure of the application program, an `XFlush(fl_get_display())` may be required following this call.

To reset the cursor to the **XForms**'s default (an arrow pointing northwest), use the following routine

```
void fl_reset_cursor(Window win)
```

To change the color of a cursor, use the following routine

```
void fl_set_cursor_color(int name, FL_COLOR fg, FL_COLOR bg)
```

where `fg` and `bg` are the foreground and background color of the cursor respectively. If the cursor is being displayed, the color change is visible immediately.

It is possible to use cursors other than those defined by the standard cursor font by creating a bitmap cursor

```
int fl_create_bitmap_cursor(const char *source, const char *mask,
                            int w, int h, int hotx, int hoty)
```

where `source` and `mask` are two (x)bitmaps. The mask defines the shape of the cursor. The pixels set to 1 in the mask defines which source pixels are displayed. If `mask` is null, all bits in source are displayed. `hotx` and `hoty` are the hotspot of the cursor (relative to the source's origin). The function returns the cursor ID which can be used in `fl_set_cursor()` and `fl_set_cursor_color()` etc.

See demo `cursor.c` for an example use of the cursor routines.

C.3 Other

As stated earlier, `fl_set_defaults()` can be used to modify **Forms Library**'s default prior to calling `fl_initialize()`. Actually this routine can also be used after `fl_initialize()` to override the values set on the command line or application databases. However, overriding users' preference should be done with discretion. Further, setting `privateColormap` after `fl_initialize()` has no effect.

Appendix D

Trouble Shooting

This appendix deals with a number of (common) problems encountered by people using the **Forms Library**. Ways of avoiding them are presented.

`fl_show_form() only draws the form partially` This only happens if immediately following `fl_show_form()`, the application program blocks the execution (e.g., waiting for a socket connection, starting a new process via `fork()` etc.). To fix this problem, you can flush the X buffer manually using `XFlush(fl_get_display())` before blocking occurs or use an idle callback to check the status of the blocking device or let the main loop handle it for you via `fl_add_io_callback()`.

`I updated the value of a slider/counter/label, but it does not change` This only happens if the update is followed by a blockage of exectuation or a long task without involving the main loop of **Forms Library**. You can force a screen update using `XSync(fl_get_display(),0)`.

`Popup menu associated with Choice object is truncated` This is a known problem and unfortunately there is no easy fix. To work around this, make sure to run your program in default visual either using command line `-visual Default`, or selecting the default visual programmatically using `fl_set_defaults()`.

Reporting Bugs

When you (think you) encountered a bug in the **XForms** please report it by sending a mail message to `zhao@bragg.phys.uwm.edu`. In this mail please indicate the version of the library, the type of machine and OS version you are running this on. The name of the window manager, and an output of `xdpyinfo` or any other relevant information (demo program similar to your code works/fails etc) would also help. **Forms Library** version can be obtained by holding the <Meta> key and pressing the middle mouse button somewhere in one of the forms, or by running `fdesign` with `-xfversion` flag. Give a short description of the problem and if possible, some sample code that exhibits the behavior. Don't expect an immediate answer but we will do our best.

Index